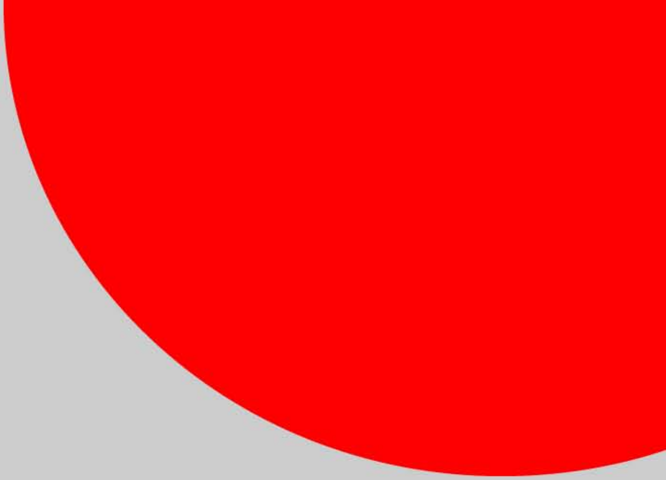
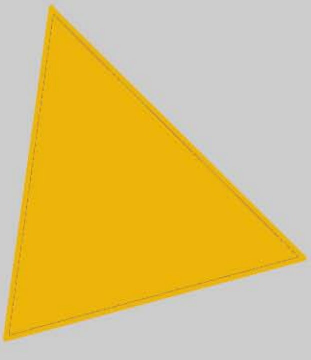


Object Pascal
Language Guide



Borland®

Object Pascal

Borland Software Corporation
100 Enterprise Way, Scotts Valley, CA 95066-3249
www.borland.com

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1983, 2002 Borland Software Corporation. All rights reserved. All Borland brand and product names are trademarks or registered trademarks of Borland Software Corporation. Other brand and product names are trademarks or registered trademarks of their respective holders.

Printed in the U.S.A.

ALP0000WW21000   1E0R0102
0203040506-9 8 7 6 5 4 3 2 1
D3

Contents

Chapter 1
Introduction 1-1

What's in this manual? 1-1
  Using Object Pascal 1-1
  Typographical conventions 1-2
Other sources of information 1-2
Software registration and technical support . . 1-3

Part I
Basic language description

Chapter 2
Overview 2-1

Program organization 2-1
  Pascal source files 2-1
  Other files used to build applications 2-2
  Compiler-generated files 2-3
Example programs. 2-3
  A simple console application 2-3
  A more complicated example 2-4
  A native application 2-5

Chapter 3
Programs and units 3-1

Program structure and syntax 3-1
  The program heading 3-2
  The program uses clause 3-2
  The block 3-2
Unit structure and syntax 3-3
  The unit heading 3-3
  The interface section 3-4
  The implementation section. 3-4
  The initialization section. 3-4
  The finalization section 3-5
Unit references and the uses clause 3-5
  The syntax of a uses clause 3-6
  Multiple and indirect unit references 3-7
  Circular unit references 3-7

Chapter 4
Syntactic elements 4-1

Fundamental syntactic elements 4-1
  Special symbols. 4-2
  Identifiers 4-2
    Qualified identifiers 4-2
  Reserved words 4-3
  Directives. 4-3
  Numerals. 4-4
  Labels 4-4
  Character strings 4-4
Comments and compiler directives. 4-5
Expressions 4-5
  Operators. 4-6
    Arithmetic operators 4-6
    Boolean operators 4-7
    Logical (bitwise) operators 4-8
    String operators 4-9
    Pointer operators. 4-9
    Set operators 4-10
    Relational operators 4-11
    Class operators. 4-12
    The @ operator 4-12
    Operator precedence rules 4-12
  Function calls 4-13
  Set constructors 4-13
  Indexes 4-14
  Typecasts 4-14
    Value typecasts. 4-14
    Variable typecasts 4-15
Declarations and statements. 4-16
  Declarations 4-16
  Statements 4-17
  Simple statements 4-17
    Assignment statements 4-17
    Procedure and function calls 4-18
    Goto statements 4-18
  Structured statements. 4-20
    Compound statements 4-20
    With statements 4-21
    If statements 4-22
    Case statements 4-24
    Control loops 4-25
    Repeat statements 4-25
    While statements. 4-26
    For statements 4-26
Blocks and scope 4-27
  Blocks. 4-28
  Scope 4-28
    Naming conflicts. 4-29

Chapter 5
Data types, variables, and constants 5-1

- About types 5-1
- Simple types 5-2
  - Ordinal types 5-2
    - Integer types 5-3
    - Character types 5-5
    - Boolean types 5-5
    - Enumerated types 5-6
    - Subrange types 5-8
  - Real types 5-9
- String types 5-10
  - Short strings 5-12
  - Long strings 5-12
  - WideString 5-13
    - About extended character sets 5-13
  - Working with null-terminated strings 5-13
    - Using pointers, arrays, and string constants 5-14
    - Mixing Pascal strings and null-terminated strings 5-15
- Structured types 5-16
  - Sets 5-17
  - Arrays 5-18
    - Static arrays 5-18
    - Dynamic arrays 5-19
    - Array types and assignments 5-21
  - Records 5-21
    - Variant parts in records 5-23
  - File types 5-25
- Pointers and pointer types 5-25
  - Overview of pointers 5-26
  - Pointer types 5-27
    - Character pointers 5-27
    - Other standard pointer types 5-28
- Procedural types 5-28
  - Procedural types in statements and expressions 5-30
- Variant types 5-31
  - Variant type conversions 5-32
  - Variants in expressions 5-33
  - Variant arrays 5-34
  - OleVariant 5-34
- Type compatibility and identity 5-35
  - Type identity 5-35
  - Type compatibility 5-36
  - Assignment-compatibility 5-36
- Declaring types 5-37
- Variables 5-37
  - Declaring variables 5-38
    - Absolute addresses 5-39
    - Dynamic variables 5-39
    - Thread-local variables 5-39
- Declared constants 5-40
  - True constants 5-40
    - Constant expressions 5-41
    - Resource strings 5-42
  - Typed constants 5-42
    - Array constants 5-43
    - Record constants 5-43
    - Procedural constants 5-44
    - Pointer constants 5-44

Chapter 6
Procedures and functions 6-1

- Declaring procedures and functions 6-1
  - Procedure declarations 6-2
  - Function declarations 6-3
  - Calling conventions 6-4
  - Forward and interface declarations 6-6
  - External declarations 6-6
    - Linking to object files 6-7
    - Importing functions from libraries 6-7
  - Overloading procedures and functions 6-8
  - Local declarations 6-10
    - Nested routines 6-11
- Parameters 6-11
  - Parameter semantics 6-12
    - Value and variable parameters 6-12
    - Constant parameters 6-13
    - Out parameters 6-13
    - Untyped parameters 6-14
  - String parameters 6-15
  - Array parameters 6-15
    - Open array parameters 6-15
    - Variant open array parameters 6-17
  - Default parameters 6-18
    - Default parameters and overloaded routines 6-19
    - Default parameters in forward and interface declarations 6-19
- Calling procedures and functions 6-19
  - Open array constructors 6-20

Chapter 7
Classes and objects 7-1

- Class types 7-2
  - Inheritance and scope 7-3

    TObject and TClass 7-3
    Compatibility of class types 7-3
    Object types 7-4
  Visibility of class members 7-4
    Private, protected, and public members . 7-5
    Published members. 7-5
    Automated members 7-6
  Forward declarations and mutually dependent classes 7-6
Fields 7-7
Methods. 7-8
  Method declarations and implementations . 7-8
    Inherited 7-9
    Self 7-9
  Method binding 7-10
    Static methods. 7-10
    Virtual and dynamic methods. 7-10
    Abstract methods 7-12
  Overloading methods 7-12
  Constructors 7-13
  Destructors 7-14
  Message methods 7-15
    Implementing message methods 7-16
    Message dispatching 7-16
Properties 7-17
  Property access 7-17
  Array properties 7-19
  Index specifiers 7-20
  Storage specifiers 7-21
  Property overrides and redeclarations. . . .7-22
Class references 7-23
  Class-reference types. 7-23
    Constructors and class references. 7-24
  Class operators 7-24
    The is operator 7-24
    The as operator 7-25
  Class methods 7-25
Exceptions 7-26
  When to use exceptions 7-26
  Declaring exception types 7-27
  Raising and handling exceptions 7-27
    Try...except statements 7-28
    Re-raising exceptions. 7-30
    Nested exceptions 7-31
    Try...finally statements 7-31
  Standard exception classes and routines. . .7-32

Chapter 8
Standard routines and I/O 8-1
File input and output. 8-1
  Text files 8-3
  Untyped files. 8-4
Text file device drivers 8-4
  Device functions. 8-5
    The Open function. 8-5
    The InOut function 8-5
    The Flush function. 8-6
    The Close function. 8-6
Handling null-terminated strings. 8-6
  Wide-character strings 8-7
Other standard routines 8-7

Part II
Special topics

Chapter 9
Libraries and packages 9-1
Calling dynamically loadable libraries 9-1
    Static loading. 9-1
    Dynamic loading. 9-2
Writing dynamically loadable libraries. 9-3
  The exports clause. 9-5
  Library initialization code 9-5
  Global variables in a library 9-6
  Libraries and system variables. 9-7
  Exceptions and runtime errors in libraries . . 9-7
  Shared-memory manager (Windows only). . 9-8
Packages 9-8
  Package declarations and source files. 9-9
    Naming packages 9-10
    The requires clause 9-10
    The contains clause 9-10
  Compiling packages 9-11
    Generated files 9-11
    Package-specific compiler directives . . 9-12
    Package-specific command-line compiler switches 9-12

Chapter 10
Object interfaces 10-1
Interface types. 10-1
  IInterface and inheritance 10-2
  Interface identification 10-3

  - Calling conventions for interfaces 10-3
  - Interface properties 10-4
  - Forward declarations 10-4
- Implementing interfaces 10-4
  - Method resolution clauses 10-5
  - Changing inherited implementations 10-6
  - Implementing interfaces by delegation 10-6
    - Delegating to an interface-type property 10-7
    - Delegating to a class-type property 10-7
- Interface references 10-8
  - Interface assignment-compatibility 10-9
  - Interface typecasts 10-10
    - Interface querying 10-10
- Automation objects (Windows only) 10-10
  - Dispatch interface types (Windows only) 10-10
    - Dispatch interface methods (Windows only) 10-11
    - Dispatch interface properties 10-11
  - Accessing Automation objects (Windows only) 10-11
    - Automation object method-call syntax 10-12
  - Dual interfaces (Windows only) 10-13

Chapter 11
Memory management 11-1
- The memory manager (Windows only) 11-1
  - Variables 11-2
- Internal data formats 11-2
  - Integer types 11-3
  - Character types 11-3
  - Boolean types 11-3
  - Enumerated types 11-3
  - Real types 11-4
    - The Real48 type 11-4
    - The Single type 11-4
    - The Double type 11-5
    - The Extended type 11-5
    - The Comp type 11-5
    - The Currency type 11-5
  - Pointer types 11-5
  - Short string types 11-5
  - Long string types 11-6
  - Wide string types 11-6
  - Set types 11-7
  - Static array types 11-7
  - Dynamic array types 11-7
  - Record types 11-8
  - File types 11-9
  - Procedural types 11-10
  - Class types 11-10
  - Class reference types 11-11
  - Variant types 11-11

Chapter 12
Program control 12-1
- Parameters and function results 12-1
  - Parameter passing 12-1
    - Register saving conventions 12-3
  - Function results 12-3
  - Method calls 12-3
    - Constructors and destructors 12-4
- Exit procedures 12-4

Chapter 13
Inline assembly code 13-1
- The asm statement 13-1
  - Register use 13-2
- Assembler statement syntax 13-2
  - Labels 13-2
  - Instruction opcodes 13-2
    - RET instruction sizing 13-3
    - Automatic jump sizing 13-3
  - Assembly directives 13-3
  - Operands 13-7
- Expressions 13-7
  - Differences between Object Pascal and assembler expressions 13-8
  - Expression elements 13-9
    - Constants 13-9
    - Registers 13-10
    - Symbols 13-10
  - Expression classes 13-12
  - Expression types 13-13
  - Expression operators 13-15
- Assembly procedures and functions 13-16

Appendix A
Object Pascal grammar A-1

Index I-1

Tables

4.1 Reserved words 4-3
4.2 Directives 4-3
4.3 Binary arithmetic operators 4-6
4.4 Unary arithmetic operators. 4-7
4.5 Boolean operators 4-7
4.6 Logical (bitwise) operators 4-8
4.7 String operators 4-9
4.8 Character-pointer operators 4-9
4.9 Set operators 4-10
4.10 Relational operators 4-11
4.11 Precedence of operators. 4-12
5.1 Generic integer types for 32-bit implementations of Object Pascal 5-3
5.2 Fundamental integer types 5-4
5.3 Fundamental real types 5-9
5.4 Generic real types 5-10
5.5 String types. 5-10
5.6 Selected pointer types declared in System and SysUtils 5-28
5.7 Variant type conversion rules 5-32
5.8 Types for integer constants 5-41
6.1 Calling conventions 6-5
8.1 Input and output procedures and functions 8-1
8.2 Null-terminated string functions 8-6
8.3 Other standard routines 8-7
9.1 Compiled package files 9-11
9.2 Package-specific compiler directives . . . 9-12
9.3 Package-specific command-line compiler switches 9-12
11.1 Long string dynamic memory layout . . 11-6
11.2 Wide string dynamic memory layout (Windows only). 11-6
11.3 Dynamic array memory layout 11-7
11.4 Type alignment masks 11-8
11.5 Virtual method table layout 11-11
13.1 Built-in assembler reserved words 13-7
13.2 String examples and their values 13-10
13.3 CPU registers 13-10
13.4 Symbols recognized by the built-in assembler 13-11
13.5 Predefined type symbols. 13-14
13.6 Precedence of built-in assembler expression operators 13-15
13.7 Definitions of built-in assembler expression operators 13-15

Chapter

1

Introduction

This manual describes the Object Pascal programming language as it is used in Borland development tools.

What's in this manual?

The first seven chapters describe most of the language elements used in ordinary programming. Chapter 8 summarizes standard routines for file I/O and string manipulation.

The next chapters describe language extensions and restrictions for dynamic-link libraries and packages (Chapter 9), and for object interfaces (Chapter 10). The final three chapters address advanced topics: memory management (Chapter 11), program control (Chapter 12), and assembly-language routines within Object Pascal programs (Chapter 13).

Using Object Pascal

The *Object Pascal Language Guide* is written to describe the Object Pascal language for use on either the Linux or Windows operating systems. Differences in the language relating to platform dependencies are noted where necessary.

Most Delphi/Kylix application developers write and compile their Object Pascal code in the integrated development environment (IDE). Working in the IDE allows the product to handle many details of setting up projects and source files, such as maintenance of dependency information among units. Borland products may enforce certain constraints on program organization that are not, strictly speaking, part of the Object Pascal language specification. For example, certain file- and program-naming conventions can be avoided if you write your programs outside of the IDE and compile them from the command prompt.

This manual generally assumes that you are working in the IDE and that you are building applications that use the Visual Component Library (VCL) and/or the Borland Component Library for Cross Platform (CLX). Occasionally, however, Borland-specific rules are distinguished from rules that apply to all Object Pascal programming.

Typographical conventions

Identifiers—that is, names of constants, variables, types, fields, properties, procedures, functions, programs, units, libraries, and packages—appear in *italics* in the text. Object Pascal operators, reserved words, and directives are in **boldface** type. Example code and text that you would type literally (into a file or at the command prompt) are in `monospaced` type.

In displayed program listings, reserved words and directives appear in **boldface**, just as they do in the text:

```
function Calculate(X, Y: Integer): Integer;
begin
  ⋮
end;
```

This is how the Code editor displays reserved words and directives, if you have the Syntax Highlight option turned on.

Some program listings, like the example above, contain ellipsis marks (... or ⋮). The ellipses represent additional code that would be included in an actual file. They are not meant to be copied literally.

In syntax descriptions, *italics* indicate placeholders for which, in real code, you would substitute syntactically valid constructions. For example, the heading of the function declaration above could be represented as

`function` *functionName*`(`*argumentList*`):` *returnType*`;`

Syntax descriptions can also contain ellipsis marks (...) and subscripts:

`function` *functionName*`(`arg_1`, ..., `arg_n`):` *ReturnType*`;`

Other sources of information

The online Help system for your development tool provides information about the IDE and user interface as well as the most up-to-date reference material for the VCL and/or CLX. Many programming topics, such as database development, are covered in depth in the *Developer's Guide*. For an overview of the documentation set, see the *Quick Start* manual that came with your software package.

Software registration and technical support

Borland Software Corporation offers a range of support plans to fit the needs of individual developers, consultants, and corporations. To receive help with this product, return the registration card and select the plan that best suits your needs. For additional information about technical support and other Borland services, contact your local sales representative or visit us online at http://www.borland.com/.

Part

I

Basic language description

The chapters in Part I present the essential language elements required for most programming tasks. These chapters include:

- Chapter 2, “Overview”
- Chapter 3, “Programs and units”
- Chapter 4, “Syntactic elements”
- Chapter 5, “Data types, variables, and constants”
- Chapter 6, “Procedures and functions”
- Chapter 7, “Classes and objects”
- Chapter 8, “Standard routines and I/O”

Chapter 2

Overview

Object Pascal is a high-level, compiled, strongly typed language that supports structured and object-oriented design. Its benefits include easy-to-read code, quick compilation, and the use of multiple unit files for modular programming.

Object Pascal has special features that support Borland's component framework and RAD environment. For the most part, descriptions and examples in this manual assume that you are using Object Pascal to develop applications using Borland development tools such as Delphi or Kylix.

Program organization

Programs are usually divided into source-code modules called *units*. Each program begins with a heading, which specifies a name for the program. The heading is followed by an optional **uses** clause, then a block of declarations and statements. The **uses** clause lists units that are linked into the program; these units, which can be shared by different programs, often have **uses** clauses of their own.

The **uses** clause provides the compiler with information about dependencies among modules. Because this information is stored in the modules themselves, Object Pascal programs do not require makefiles, header files, or preprocessor "include" directives. (The Project Manager generates a makefile each time a project is loaded in the IDE, but saves these files only for project groups that include more than one project.)

For further discussion of program structure and dependencies, see Chapter 3, "Programs and units".

Pascal source files

The compiler expects to find Pascal source code in files of three kinds:

- *unit* source files (which end with the .pas extension)

- *project* files (which end with the .dpr extension)
- *package* source files (which end with the .dpk extension)

Unit source files contain most of the code in an application. Each application has a single project file and several unit files; the project file—which corresponds to the "main" program file in traditional Pascal—organizes the unit files into an application. Borland development tools automatically maintain a project file for each application.

If you are compiling a program from the command line, you can put all your source code into unit (.pas) files. But if you use the IDE to build your application, you must have a project (.dpr) file.

Package source files are similar to project files, but they are used to construct special dynamically linkable libraries called *packages*. For more information about packages, see Chapter 9, "Libraries and packages".

Other files used to build applications

In addition to source-code modules, Borland products use several non-Pascal files to build applications. These files are maintained automatically and include

- *form* files, which end with the .dfm (Delphi) or .xfm (Kylix) extension,
- *resource* files, which end with the .res extension, and
- *project options* files, which end with the .dof (Delphi) or .kof (Kylix) extension.

A form file is either a text file or a compiled resource file that can contain bitmaps, strings, and so forth. Each form file represents a single form, which usually corresponds to a window or dialog box in an application. The IDE allows you to view and edit form files as text, and to save form files as either text or binary. Although the default behavior is to save form files as text, they are usually not edited manually; it is more common to use Borland's visual design tools for this purpose. Each project has at least one form, and each form has an associated unit (.pas) file that, by default, has the same name as the form file.

In addition to form files, each project uses a resource (.res) file to hold the bitmap for the application's icon. By default, this file has the same name as the project (.dpr) file. To change an application's icon, use the Project Options dialog.

A project options (.dof or .kof) file contains compiler and linker settings, search directories, version information, and so forth. Each project has an associated project options file with the same name as the project (.dpr) file. Usually, the options in this file are set from Project Options dialog.

Various tools in the IDE store data in files of other types. Desktop settings (.dsk or .desk) files contain information about the arrangement of windows and other configuration options; desktop settings can be project-specific or environment-wide. These files have no direct effect on compilation.

Compiler-generated files

The first time you build an application or a standard dynamic-link library, the compiler produces a compiled unit .dcu (Windows) .dcu/.dpu (Linux) file for each new unit used in your project; all the .dcu (Windows) .dcu/.dpu (Linux) files in your project are then linked to create a single executable or shared library file. The first time you build a package, the compiler produces a .dcu (Windows) .dpu (Linux) file for each new unit contained in the package, and then creates both a .dcp and a package file. (For more information about libraries and packages, see Chapter 9.) If you use the **–GD** switch, the linker generates a map file and a .drc file; the .drc file, which contains string resources, can be compiled into a resource file.

When you rebuild a project, individual units are not recompiled unless their source (.pas) files have changed since the last compilation, or their .dcu (Windows) .dcu/ .dpu (Linux) files cannot be found, or you explicitly tell the compiler to reprocess them. In fact, it is not necessary for a unit's source file to be present at all, as long as the compiler can find the compiled unit file.

Example programs

The examples that follow illustrate basic features of Object Pascal programming. The examples show simple Object Pascal applications that cannot be compiled from the IDE; but you can compile them from the command line.

A simple console application

The program below is a simple console application that you can compile and run from the command prompt.

```
program Greeting;

{$APPTYPE CONSOLE}

var MyMessage: string;

begin
  MyMessage := 'Hello world!';
  Writeln(MyMessage);
end.
```

The first line declares a program called *Greeting*. The `{$APPTYPE CONSOLE}` directive tells the compiler that this is a console application, to be run from the command line. The next line declares a variable called *MyMessage*, which holds a string. (Object Pascal has genuine string data types.) The program then assigns the string "Hello world!" to the variable *MyMessage*, and sends the contents of *MyMessage* to the standard output using the *Writeln* procedure. (*Writeln* is defined implicitly in the *System* unit, which the compiler automatically includes in every application.)

You can type this program into a file called Greeting.pas or Greeting.dpr and compile it by entering

```
On Delphi: DCC32 Greeting
```

```
On Kylix: dcc Greeting
```

on the command line. The resulting executable prints the message "Hello world!"

Aside from its simplicity, this example differs in several important ways from programs that you are likely to write with Borland development tools. First, it is a console application. Borland development tools are typically used to write applications with graphical interfaces; hence, you would not ordinarily call *Writeln*. Moreover, the entire example program (save for *Writeln*) is in a single file. In a typical application, the program heading—the first line of the example—would be placed in a separate project file that would not contain any of the actual application logic, other than a few calls to *methods* defined in unit files.

A more complicated example

The next example shows a program that is divided into two files: a *project* file and a *unit* file. The project file, which you can save as Greeting.dpr, looks like this:

```
program Greeting;

{$APPTYPE CONSOLE}

uses Unit1;

begin
  PrintMessage('Hello World!');
end.
```

The first line declares a program called *Greeting*, which, once again, is a console application. The `uses Unit1;` clause tells the compiler that *Greeting* includes a unit called *Unit1*. Finally, the program calls the *PrintMessage* procedure, passing to it the string "Hello World!" Where does the *PrintMessage* procedure come from? It's defined in *Unit1*. Here's the source code for *Unit1*, which you can save in a file called Unit1.pas:

```
unit Unit1;

interface

procedure PrintMessage(msg: string);

implementation

procedure PrintMessage(msg: string);
begin
  Writeln(msg);
end;

end.
```

Unit1 defines a procedure called *PrintMessage* that takes a single string as an argument and sends the string to the standard output. (In Pascal, routines that do not return a value are called *procedures*. Routines that return a value are called *functions*.) Notice that *PrintMessage* is declared twice in *Unit1*. The first declaration, under the reserved word **interface**, makes *PrintMessage* available to other modules (such as *Greeting*) that use *Unit1*. The second declaration, under the reserved word **implementation**, actually defines *PrintMessage*.

You can now compile *Greeting* from the command line by entering

```
On Delphi: DCC32 Greeting
```

```
On Kylix: dcc Greeting
```

There's no need to include *Unit1* as a command-line argument. When the compiler processes Greeting.dpr, it automatically looks for unit files that the *Greeting* program depends on. The resulting executable does the same thing as our first example: it prints the message "Hello world!"

A native application

Our next example is an application built using VCL or CLX components in the IDE. This program uses automatically generated form and resource files, so you won't be able to compile it from the source code alone. But it illustrates important features of Object Pascal. In addition to multiple units, the program uses classes and objects, which are discussed in Chapter 7, "Classes and objects".

The program includes a project file and two new unit files. First, the project file:

```
program Greeting;  { comments are enclosed in braces }

uses
  Forms, {change the unit name to QForms on Linux}
  Unit1 in 'Unit1.pas' { the unit for Form1 },
  Unit2 in 'Unit2.pas' { the unit for Form2 };

{$R *.res}  { this directive links the project's resource file }

begin
  { calls to Application }
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

Once again, our program is called *Greeting*. It uses three units: *Forms*, which is part of VCL and CLX; *Unit1*, which is associated with the application's main form (*Form1*); and *Unit2*, which is associated with another form (*Form2*).

The program makes a series of calls to an object named *Application*, which is an instance of the *TApplication* class defined in the *Forms* unit. (Every project has an automatically generated *Application* object.) Two of these calls invoke a *TApplication* method named *CreateForm*. The first call to *CreateForm* creates *Form1*, an instance of the *TForm1* class defined in *Unit1*. The second call to *CreateForm* creates *Form2*, an instance of the *TForm2* class defined in *Unit2*.

Unit1 looks like this:

```
unit Unit1;

interface

uses  { these units are part of the Visual Component Library (VCL) }
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;

{
On Linux, the uses clause looks like this:
uses  { these units are part of CLX }
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
}

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

uses Unit2;  { this is where Form2 is defined }

{$R *.dfm}  { this directive links Unit1's form file }

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.ShowModal;
end;

end.
```

Unit1 creates a class named *TForm1* (derived from *TForm*) and an instance of this class, *Form1*. *TForm1* includes a button—*Button1*, an instance of *TButton*—and a procedure named *TForm1.Button1Click* that is called at runtime whenever the user presses *Button1*. *TForm1.Button1Click* hides *Form1* and it displays *Form2* (the call to *Form2.ShowModal*). *Form2* is defined in *Unit2*:

```
unit Unit2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
```

```
{
On Linux, the uses clause looks like this:
uses  { these units are part of CLX }
  SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
}
type
  TForm2 = class(TForm)
    Label1: TLabel;
    CancelButton: TButton;
    procedure CancelButtonClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  end;

var
  Form2: TForm2;

implementation

uses Unit1;

{$R *.dfm}

procedure TForm2.CancelButtonClick(Sender: TObject);
begin
  Form2.Close;
end;

end.
```

Unit2 creates a class named *TForm2* and an instance of this class, *Form2*. *TForm2* includes a button (*CancelButton*, an instance of *TButton*) and a label (*Label1*, an instance of *TLabel*). You can't see this from the source code, but *Label1* displays a caption that reads "Hello world!" The caption is defined in *Form2*'s form file, Unit2.dfm.

Unit2 defines one procedure. *TForm2.CancelButtonClick* is called at runtime whenever the user presses *CancelButton*; it closes *Form2*. This procedure (along with *Unit1*'s *TForm1.Button1Click*) is known as an *event handler* because it responds to events that occur while the program is running. Event handlers are assigned to specific events by the form (.dfm on Windows .xfm on Linux) files for *Form1* and *Form2*.

When the *Greeting* program starts, *Form1* is displayed and *Form2* is invisible. (By default, only the first form created in the project file is visible at runtime. This is called the project's *main form*.) When the user presses the button on *Form1*, *Form1* disappears and is replaced by *Form2*, which displays the "Hello world!" greeting. When the user closes *Form2* (by pressing *CancelButton* or the Close button on the title bar), *Form1* reappears.

Chapter

3

Programs and units

A program is constructed from source-code modules called *units*. Each unit is stored in its own file and compiled separately; compiled units are linked to create an application. Units allow you to

- divide large programs into modules that can be edited separately.
- create libraries that you can share among programs.
- distribute libraries to other developers without making the source code available.

In traditional Pascal programming, all source code, including the main program, is stored in .pas files. Borland tools use a *project* (.dpr) file to store the "main" program, while most other source code resides in *unit* (.pas) files. Each application—or *project*—consists of a single project file and one or more unit files. (Strictly speaking, you needn't explicitly use any units in a project, but all programs automatically use the *System* unit.) To build a project, the compiler needs either a source file or a compiled unit file for each unit.

Program structure and syntax

A program contains

- a program heading,
- a **uses** clause (optional), and
- a block of declarations and statements.

The program heading specifies a name for the program. The **uses** clause lists units used by the program. The block contains declarations and statements that are executed when the program runs. The IDE expects to find these three elements in a single project (.dpr) file.

The example below shows the project file for a program called Editor.

```
 1    program Editor;
 2
 3    uses
 4      Forms, {change to QForms in Linux}
 5      REAbout in 'REAbout.pas' {AboutBox},
 6      REMain in 'REMain.pas' {MainForm};
 7
 8    {$R *.res}
 9
10    begin
11      Application.Title := 'Text Editor';
12      Application.CreateForm(TMainForm, MainForm);
13      Application.Run;
14    end.
```

Line 1 contains the program heading. The **uses** clause is on lines 3 through 6. Line 8 is a compiler directive that links the project's resource file into the program. Lines 10 through 14 contain the block of statements that are executed when the program runs. Finally, the project file, like all source files, ends with a period.

This is, in fact, a fairly typical project file. Project files are usually short, since most of a program's logic resides in its unit files. Project files are generated and maintained automatically, and it is seldom necessary to edit them manually.

The program heading

The program heading specifies the program's name. It consists of the reserved word **program**, followed by a valid identifier, followed by a semicolon. The identifier must match the project file name. In the example above, since the program is called Editor, the project file should be called EDITOR.dpr.

In standard Pascal, a program heading can include parameters after the program name:

```
program Calc(input, output);
```

Borland's Object Pascal compiler ignores these parameters.

The program uses clause

The **uses** clause lists units that are incorporated into the program. These units may in turn have **uses** clauses of their own. For more information about the **uses** clause, see "Unit references and the uses clause" on page 3-5.

The block

The block contains a simple or structured statement that is executed when the program runs. In most programs, the block consists of a compound statement—bracketed between the reserved words **begin** and **end**—whose component statements are simply method calls to the project's *Application* object. (Every project has an *Application* variable that holds an instance of *TApplication*, *TWebApplication*, or *TServiceApplication*.) The block can also contain declarations of constants, types, variables, procedures, and functions; these declarations must precede the statement part of the block.

Unit structure and syntax

A unit consists of types (including classes), constants, variables, and routines (functions and procedures). Each unit is defined in its own unit (.pas) file.

A unit file begins with a unit heading, which is followed by the *interface*, *implementation*, *initialization*, and *finalization* sections. The initialization and finalization sections are optional. A skeleton unit file looks like this:

```
unit Unit1;

interface

uses  { List of units goes here }

  { Interface section goes here }

implementation

uses  { List of units goes here }

  { Implementation section goes here }

initialization
  { Initialization section goes here }

finalization
  { Finalization section goes here }

end.
```

The unit must conclude with the word **end** followed by a period.

The unit heading

The unit heading specifies the unit's name. It consists of the reserved word **unit**, followed by a valid identifier, followed by a semicolon. For applications developed using Borland tools, the identifier must match the unit file name. Thus, the unit heading

```
unit MainForm;
```

would occur in a source file called MAINFORM.pas, and the file containing the compiled unit would be MAINFORM.dcu.

Unit names must be unique within a project. Even if their unit files are in different directories, two units with the same name cannot be used in a single program.

The interface section

The interface section of a unit begins with the reserved word **interface** and continues until the beginning of the implementation section. The interface section declares constants, types, variables, procedures, and functions that are available to *clients*—that is, to other units or programs that use the unit where they are declared. These entities are called *public* because a client can access them as if they were declared in the client itself.

The interface declaration of a procedure or function includes only the routine's heading. The block of the procedure or function follows in the implementation section. Thus procedure and function declarations in the interface section work like forward declarations, although the **forward** directive isn't used.

The interface declaration for a class must include declarations for all class members.

The interface section can include its own **uses** clause, which must appear immediately after the word **interface**. For information about the **uses** clause, see "Unit references and the uses clause" on page 3-5.

The implementation section

The implementation section of a unit begins with the reserved word **implementation** and continues until the beginning of the initialization section or, if there is no initialization section, until the end of the unit. The implementation section defines procedures and functions that are declared in the interface section. Within the implementation section, these procedures and functions may be defined and called in any order. You can omit parameter lists from public procedure and function headings when you define them in the implementation section; but if you include a parameter list, it must match the declaration in the interface section exactly.

In addition to definitions of public procedures and functions, the implementation section can declare constants, types (including classes), variables, procedures, and functions that are *private* to the unit—that is, inaccessible to clients.

The implementation section can include its own **uses** clause, which must appear immediately after the word **implementation**. For information about the **uses** clause, see "Unit references and the uses clause" on page 3-5.

The initialization section

The initialization section is optional. It begins with the reserved word **initialization** and continues until the beginning of the finalization section or, if there is no finalization section, until the end of the unit. The initialization section contains statements that are executed, in the order in which they appear, on program start-up.

So, for example, if you have defined data structures that need to be initialized, you can do this in the initialization section.

The initialization sections of units used by a client are executed in the order in which the units appear in the client's **uses** clause.

The finalization section

The finalization section is optional and can appear only in units that have an initialization section. The finalization section begins with the reserved word **finalization** and continues until the end of the unit. It contains statements that are executed when the main program terminates. Use the finalization section to free resources that are allocated in the initialization section.

Finalization sections are executed in the opposite order from initializations. For example, if your application initializes units *A*, *B*, and *C*, in that order, it will finalize them in the order *C*, *B*, and *A*.

Once a unit's initialization code starts to execute, the corresponding finalization section is guaranteed to execute when the application shuts down. The finalization section must therefore be able to handle incompletely initialized data, since, if a runtime error occurs, the initialization code might not execute completely.

Unit references and the uses clause

A **uses** clause lists units used by the program, library, or unit in which the clause appears. (For information about libraries, see Chapter 9, "Libraries and packages".) A **uses** clause can occur in

- the project file for a program or library,
- the interface section of a unit, and
- the implementation section of a unit.

Most project files contain a **uses** clause, as do the interface sections of most units. The implementation section of a unit can contain its own **uses** clause as well.

The *System* unit is used automatically by every application and cannot be listed explicitly in the **uses** clause. (*System* implements routines for file I/O, string handling, floating point operations, dynamic memory allocation, and so forth.) Other standard library units, such as *SysUtils*, must be included in the **uses** clause. In most cases, all necessary units are placed in the **uses** clause when your project generates and maintains a source file.

In unit declarations and **uses** clauses (on Kylix particularly), unit names must match the file names in case. In other contexts (such as qualified identifiers), unit names are case insensitive. To avoid problems with unit references, refer to the unit source file explicitly:

```
uses MyUnit in "myunit.pas";
```

If such an explicit reference appears in the project file, other source files can refer to the unit with a simple **uses** clause that does not need to match case:

```
uses Myunit;
```

For more information about the placement and content of the **uses** clause, see "Multiple and indirect unit references" on page 3-7 and "Circular unit references" on page 3-7.

The syntax of a uses clause

A **uses** clause consists of the reserved word **uses**, followed by one or more comma-delimited unit names, followed by a semicolon. Examples:

```
uses Forms, Main;

uses Windows, Messages, SysUtils, Strings, Classes, Unit2, MyUnit;

uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
```

In the **uses** clause of a program or library, any unit name may be followed by the reserved word **in** and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. Examples:

```
uses Windows, Messages, SysUtils, Strings in 'C:\Classes\Strings.pas', Classes;

uses
  QForms,
  Main,
  Extra in '../extra/extra.pas';
```

Include **in** ... after a unit name when you need to specify the unit's source file. Since the IDE expects unit names to match the names of the source files in which they reside, there is usually no reason to do this. Using **in** is necessary only when the location of the source file is unclear, for example when

- You have used a source file that is in a different directory from the project file, and that directory is not in the compiler's search path or the general Library search path.
- Different directories in the compiler's search path have identically named units.
- You are compiling a console application from the command line, and you have named a unit with an identifier that doesn't match the name of its source file.

The compiler also relies on the **in** ... construction to determine which units are part of a project. Only units that appear in a project (.dpr) file's **uses** clause followed by **in** and a file name are considered to be part of the project; other units in the **uses** clause are used by the project without belonging to it. This distinction has no effect on compilation, but it affects IDE tools like the Project Manager and Project Browser.

In the **uses** clause of a unit, you cannot use **in** to tell the compiler where to find a source file. Every unit must be in the compiler's search path, the general Library search path, or the same directory as the unit that uses it. Moreover, unit names must match the names of their source files.

Multiple and indirect unit references

The order in which units appear in the **uses** clause determines the order of their initialization (see "The initialization section" on page 3-4) and affects the way identifiers are located by the compiler. If two units declare a variable, constant, type, procedure, or function with the same name, the compiler uses the one from the unit listed last in the **uses** clause. (To access the identifier from the other unit, you would have to add a qualifier: *UnitName.Identifier*.)

A **uses** clause need include only units used directly by the program or unit in which the clause appears. That is, if unit *A* references constants, types, variables, procedures, or functions that are declared in unit *B*, then *A* must use *B* explicitly. If *B* in turn references identifiers from unit *C*, then *A* is indirectly dependent on *C*; in this case, *C* needn't be included in a **uses** clause in *A*, but the compiler must still be able to find both *B* and *C* in order to process *A*.

The example below illustrates indirect dependency.

```
program Prog;
uses Unit2;
const a = b;
⋮
unit Unit2;
interface
uses Unit1;
const b = c;
⋮
unit Unit1;
interface
const c = 1;
⋮
```

In this example, *Prog* depends directly on *Unit2*, which depends directly on *Unit1*. Hence *Prog* is indirectly dependent on *Unit1*. Because *Unit1* does not appear in *Prog*'s **uses** clause, identifiers declared in *Unit1* are not available to *Prog*.

To compile a client module, the compiler needs to locate all units that the client depends on, directly or indirectly. Unless the source code for these units has changed, however, the compiler needs only their .dcu (Windows) or .dcu/.dpu (Linux) files, not their source (.pas) files.

When changes are made in the interface section of a unit, other units that depend on it must be recompiled. But when changes are made only in the implementation or other sections of a unit, dependent units don't have to be recompiled. The compiler tracks these dependencies automatically and recompiles units only when necessary.

Circular unit references

When units reference each other directly or indirectly, the units are said to be mutually dependent. Mutual dependencies are allowed as long as there are no circular paths connecting the **uses** clause of one interface section to the **uses** clause of another. In other words, starting from the interface section of a unit, it must never be possible to return to that unit by following references through interface sections of other units. For a pattern of mutual dependencies to be valid, each circular reference path must lead through the **uses** clause of at least one implementation section.

In the simplest case of two mutually dependent units, this means that the units cannot list each other in their interface **uses** clauses. So the following example leads to a compilation error:

```
unit Unit1;
interface
uses Unit2;
⋮

unit Unit2;
interface
uses Unit1;
⋮
```

However, the two units can legally reference each other if one of the references is moved to the implementation section:

```
unit Unit1;
interface
uses Unit2;
⋮

unit Unit2;
interface
⋮
implementation
uses Unit1;
⋮
```

To reduce the chance of circular references, it’s a good idea to list units in the implementation **uses** clause whenever possible. Only when identifiers from another unit are used in the interface section is it necessary to list that unit in the interface **uses** clause.

Chapter

4

Syntactic elements

Object Pascal uses the ASCII character set, including the letters *A* through *Z* and *a* through *z*, the digits *0* through *9*, and other standard characters. It is *not* case-sensitive. The space character (ASCII 32) and the control characters (ASCII 0 through 31—including ASCII 13, the return or end-of-line character) are called *blanks*.

Fundamental syntactic elements, called *tokens*, combine to form expressions, declarations, and statements. A *statement* describes an algorithmic action that can be executed within a program. An *expression* is a syntactic unit that occurs within a statement and denotes a value. A *declaration* defines an identifier (such as the name of a function or variable) that can be used in expressions and statements, and, where appropriate, allocates memory for the identifier.

Fundamental syntactic elements

On the simplest level, a program is a sequence of tokens delimited by separators. A *token* is the smallest meaningful unit of text in a program. A *separator* is either a blank or a comment. Strictly speaking, it is not always necessary to place a separator between two tokens; for example, the code fragment

```
Size:=20;Price:=10;
```

is perfectly legal. Convention and readability, however, dictate that we write this as

```
Size := 20;
Price := 10;
```

Tokens are categorized as *special symbols*, *identifiers*, *reserved words*, *directives*, *numerals*, *labels*, and *character strings*. A separator can be part of a token only if the token is a character string. Adjacent identifiers, reserved words, numerals, and labels must have one or more separators between them.

Special symbols

Special symbols are non-alphanumeric characters, or pairs of such characters, that have fixed meanings. The following single characters are special symbols.

# $ & ' ( ) * + , − . / : ; < = > @ [ ] ^ { }

The following character pairs are also special symbols.

(* (. *) .) .. // := <= >= <>

The left bracket—**[**—is equivalent to the character pair of left parenthesis and period—**(.**; the right bracket—**]**—is equivalent to the character pair of period and right parenthesis—**.)** . The left-parenthesis–plus–asterisk and asterisk–plus–right-parenthesis—**(* *)**—are equivalent to the left and right brace—**{ }**.

Notice that **!**, **"** (double quotation marks), **%**, **?**, ****, **_** (underscore), **|** (pipe), and **~** (tilde) are not special characters.

Identifiers

Identifiers denote constants, variables, fields, types, properties, procedures, functions, programs, units, libraries, and packages. An identifier can be of any length, but only the first 255 characters are significant. An identifier must begin with a letter or an underscore (_) and cannot contain spaces; letters, digits, and underscores are allowed after the first character. Reserved words cannot be used as identifiers.

Since Object Pascal is case-insensitive, an identifier like *CalculateValue* could be written in any of these ways:

```
CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE
```

On Linux, the only identifiers for which case is important are unit names. Since unit names correspond to file names, inconsistencies in case can sometimes affect compilation.

Qualified identifiers

When you use an identifier that has been declared in more than one place, it is sometimes necessary to *qualify* the identifier. The syntax for a qualified identifier is

identifier$_1$.*identifier*$_2$

where *identifier*$_1$ qualifies *identifier*$_2$. For example, if two units each declare a variable called *CurrentValue*, you can specify that you want to access the *CurrentValue* in *Unit2* by writing

```
Unit2.CurrentValue
```

Qualifiers can be iterated. For example,

```
Form1.Button1.Click
```

calls the *Click* method in *Button1* of *Form1*.

If you don't qualify an identifier, its interpretation is determined by the rules of scope described in "Blocks and scope" on page 4-27.

Reserved words

The following reserved words cannot be redefined or used as identifiers.

Table 4.1 Reserved words

and	**downto**	**in**	**or**	**string**
array	**else**	**inherited**	**out**	**then**
as	**end**	**initialization**	**packed**	**threadvar**
asm	**except**	**inline**	**procedure**	**to**
begin	**exports**	**interface**	**program**	**try**
case	**file**	**is**	**property**	**type**
class	**finalization**	**label**	**raise**	**unit**
const	**finally**	**library**	**record**	**until**
constructor	**for**	**mod**	**repeat**	**uses**
destructor	**function**	**nil**	**resourcestring**	**var**
dispinterface	**goto**	**not**	**set**	**while**
div	**if**	**object**	**shl**	**with**
do	**implementation**	**of**	**shr**	**xor**

In addition to the words in Table 4.1, **private**, **protected**, **public**, **published**, and **automated** act as reserved words within object type declarations, but are otherwise treated as directives. The words **at** and **on** also have special meanings.

Directives

Directives are words that are sensitive in specific locations within source code. Directives have special meanings in Object Pascal, but, unlike reserved words, appear only in contexts where user-defined identifiers cannot occur. Hence—although it is inadvisable to do so—you can define an identifier that looks exactly like a directive.

Table 4.2 Directives

absolute	**dynamic**	**message**	**private**	**resident**
abstract	**export**	**name**	**protected**	**safecall**
assembler	**external**	**near**	**public**	**stdcall**
automated	**far**	**nodefault**	**published**	**stored**
cdecl	**forward**	**overload**	**read**	**varargs**
contains	**implements**	**override**	**readonly**	**virtual**
default	**index**	**package**	**register**	**write**
deprecated	**library**	**pascal**	**reintroduce**	**writeonly**
dispid	**local**	**platform**	**requires**	

Numerals

Integer and real constants can be represented in decimal notation as sequences of digits without commas or spaces, and prefixed with the **+** or **–** operator to indicate sign. Values default to positive (so that, for example, `67258` is equivalent to `+67258`) and must be within the range of the largest predefined real or integer type.

Numerals with decimal points or exponents denote reals, while other numerals denote integers. When the character *E* or *e* occurs within a real, it means "times ten to the power of". For example, `7E-2` means 7×10^{-2}, and `12.25e+6` and `12.25e6` both mean 12.25×10^{6}.

The dollar-sign prefix indicates a hexadecimal numeral—for example, `$8F`. Hexadecimal numbers without a preceding '-' unary operator are taken to be positive values. During an assignment, if a hexadecimal value lies outside the range of the receiving type an error is raised, except in the case of the Integer (32-bit Integer) where a warning is raised. In this case, values exceeding the positive range for Integer are taken to be negative numbers in a manner consistent with 2's complement integer representation.

For more information about real and integer types, see Chapter 5, "Data types, variables, and constants". For information about the data types of numerals, see "True constants" on page 5-40.

Labels

A label is a standard Object Pascal identifier with the exception that, unlike other identifiers, labels can start with a digit. Numeric labels can include no more than ten digits—that is, a numeral between 0 and 9999999999.

Labels are used in **goto** statements. For more information about **goto** statements and labels, see "Goto statements" on page 4-18.

Character strings

A character string, also called a *string literal* or *string constant*, consists of a *quoted string*, a *control string*, or a combination of quoted and control strings. Separators can occur only within quoted strings.

A quoted string is a sequence of up to 255 characters from the extended ASCII character set, written on one line and enclosed by apostrophes. A quoted string with nothing between the apostrophes is a *null string*. Two sequential apostrophes in a quoted string denote a single character, namely an apostrophe. For example,

```
'BORLAND'          { BORLAND }
'You''ll see'      { You'll see }
''''               { ' }
''                 { null string }
' '                { a space }
```

A control string is a sequence of one or more *control characters*, each of which consists of the **#** symbol followed by an unsigned integer constant from 0 to 255 (decimal or hexadecimal) and denotes the corresponding ASCII character. The control string

```
#89#111#117
```

is equivalent to the quoted string

```
'You'
```

You can combine quoted strings with control strings to form larger character strings. For example, you could use

```
'Line 1'#13#10'Line 2'
```

to put a carriage-return–line-feed between "Line 1" and "Line 2". However, you cannot concatenate two quoted strings in this way, since a pair of sequential apostrophes is interpreted as a single character. (To concatenate quoted strings, use the + operator described in "String operators" on page 4-9, or simply combine them into a single quoted string.)

A character string's *length* is the number of characters in the string. A character string of any length is compatible with any string type and with the *PChar* type. A character string of length 1 is compatible with any character type, and, when extended syntax is enabled (**{$X+}**), a character string of length $n \geq 1$ is compatible with zero-based arrays and packed arrays of n characters. For more information about string types, see Chapter 5, "Data types, variables, and constants".

Comments and compiler directives

Comments are ignored by the compiler, except when they function as separators (delimiting adjacent tokens) or compiler directives.

There are several ways to construct comments:

```
{ Text between a left brace and a right brace constitutes a comment. }

(* Text between a left-parenthesis-plus-asterisk and an
   asterisk-plus-right-parenthesis also constitutes a comment. *)

// Any text between a double-slash and the end of the line constitutes a comment.
```

A comment that contains a dollar sign (**$**) immediately after the opening **{** or **(*** is a compiler directive. For example,

```
{$WARNINGS OFF}
```

tells the compiler not to generate warning messages.

Expressions

An *expression* is a construction that returns a value. For example,

```
X                    { variable }
@X                   { address of a variable }
15                   { integer constant }
```

```
InterestRate        { variable }
Calc(X,Y)           { function call }
X * Y               { product of X and Y }
Z / (1 - Z)         { quotient of Z and (1 - Z) }
X = 1.5             { Boolean }
C in Range1         { Boolean }
not Done            { negation of a Boolean }
['a','b','c']       { set }
Char(48)            { value typecast }
```

The simplest expressions are variables and constants (described in Chapter 5, "Data types, variables, and constants"). More complex expressions are built from simpler ones using *operators*, *function calls*, *set constructors*, *indexes*, and *typecasts*.

Operators

Operators behave like predefined functions that are part of the Object Pascal language. For example, the expression `(X + Y)` is built from the variables `X` and `Y`—called *operands*—with the **+** operator; when `X` and `Y` represent integers or reals, `(X + Y)` returns their sum. Operators include **@**, **not**, **^**, *****, **/**, **div**, **mod**, **and**, **shl**, **shr**, **as**, **+**, **–**, **or**, **xor**, **=**, **>**, **<**, **<>**, **<=**, **>=**, **in**, and **is**.

The operators **@**, **not**, and **^** are *unary* (taking one operand). All other operators are *binary* (taking two operands), except that **+** and **–** can function as either unary or binary. A unary operator always precedes its operand (for example, `-B`), except for **^**, which follows its operand (for example, `P^`). A binary operator is placed between its operands (for example, `A = 7`).

Some operators behave differently depending on the type of data passed to them. For example, **not** performs bitwise negation on an integer operand and logical negation on a Boolean operand. Such operators appear below under multiple categories.

Except for **^**, **is**, and **in**, all operators can take operands of type *Variant*. For details, see "Variant types" on page 5-31.

The sections that follow assume some familiarity with Object Pascal data types. For information about data types, see Chapter 5, "Data types, variables, and constants".

For information about operator precedence in complex expressions, see "Operator precedence rules" on page 4-12.

Arithmetic operators

Arithmetic operators, which take real or integer operands, include **+**, **–**, *****, **/**, **div**, and **mod**.

Table 4.3 Binary arithmetic operators

Operator	Operation	Operand types	Result type	Example
+	addition	integer, real	integer, real	`X + Y`
–	subtraction	integer, real	integer, real	`Result - 1`
*	multiplication	integer, real	integer, real	`P * InterestRate`

Table 4.3 Binary arithmetic operators (continued)

Operator	Operation	Operand types	Result type	Example
/	real division	integer, real	real	`X / 2`
div	integer division	integer	integer	`Total` **div** `UnitSize`
mod	remainder	integer	integer	`Y` **mod** `6`

Table 4.4 Unary arithmetic operators

Operator	Operation	Operand type	Result type	Example
+	sign identity	integer, real	integer, real	`+7`
–	sign negation	integer, real	integer, real	`-X`

The following rules apply to arithmetic operators.

- The value of *x*/*y* is of type *Extended*, regardless of the types of *x* and *y*. For other arithmetic operators, the result is of type *Extended* whenever at least one operand is a real; otherwise, the result is of type *Int64* when at least one operand is of type *Int64*; otherwise, the result is of type *Integer*. If an operand's type is a subrange of an integer type, it is treated as if it were of the integer type.
- The value of *x* **div** *y* is the value of *x*/*y* rounded in the direction of zero to the nearest integer.
- The **mod** operator returns the remainder obtained by dividing its operands. In other words, *x* **mod** *y* = *x* – (*x* **div** *y*) * *y*.
- A runtime error occurs when *y* is zero in an expression of the form *x*/*y*, *x* **div** *y*, or *x* **mod** *y*.

Boolean operators

The Boolean operators **not**, **and**, **or**, and **xor** take operands of any Boolean type and return a value of type *Boolean*.

Table 4.5 Boolean operators

Operator	Operation	Operand types	Result type	Example
not	negation	Boolean	*Boolean*	**not** `(C` **in** `MySet)`
and	conjunction	Boolean	*Boolean*	`Done` **and** `(Total > 0)`
or	disjunction	Boolean	*Boolean*	`A` **or** `B`
xor	exclusive disjunction	Boolean	*Boolean*	`A` **xor** `B`

These operations are governed by standard rules of Boolean logic. For example, an expression of the form *x* **and** *y* is *True* if and only if both *x* and *y* are *True*.

Complete versus short-circuit Boolean evaluation

The compiler supports two modes of evaluation for the **and** and **or** operators: complete evaluation and short-circuit (partial) evaluation. *Complete evaluation* means that each conjunct or disjunct is evaluated, even when the result of the entire expression is already determined. *Short-circuit evaluation* means strict left-to-right evaluation that stops as soon as the result of the entire expression is determined. For example, if the expression `A and B` is evaluated under short-circuit mode when *A* is *False*, the compiler won't evaluate *B*; it knows that the entire expression is *False* as soon as it evaluates *A*.

Short-circuit evaluation is usually preferable because it guarantees minimum execution time and, in most cases, minimum code size. Complete evaluation is sometimes convenient when one operand is a function with side effects that alter the execution of the program.

Short-circuit evaluation also allows the use of constructions that might otherwise result in illegal runtime operations. For example, the following code iterates through the string *S*, up to the first comma.

```
while (I <= Length(S)) and (S[I] <> ',') do
begin
  ⋮
  Inc(I);
end;
```

In a case where *S* has no commas, the last iteration increments *I* to a value which is greater than the length of *S*. When the **while** condition is next tested, complete evaluation results in an attempt to read *S[I]*, which could cause a runtime error. Under short-circuit evaluation, in contrast, the second part of the **while** condition—`(S[I] <> ',')`—is not evaluated after the first part fails.

Use the **$B** compiler directive to control evaluation mode. The default state is **{$B–}**, which enables short-circuit evaluation. To enable complete evaluation locally, add the **{$B+}** directive to your code. You can also switch to complete evaluation on a project-wide basis by selecting Complete Boolean Evaluation in the Compiler Options dialog.

Note If either operand involves a variant, the compiler always performs complete evaluation (even in the **{$B–}** state).

Logical (bitwise) operators

The following logical operators perform bitwise manipulation on integer operands. For example, if the value stored in *X* (in binary) is 001101 and the value stored in *Y* is 100001, the statement

```
Z := X or Y;
```

assigns the value 101101 to *Z*.

Table 4.6 Logical (bitwise) operators

Operator	Operation	Operand types	Result type	Examples
not	bitwise negation	integer	integer	**not** `X`
and	bitwise and	integer	integer	`X` **and** `Y`
or	bitwise or	integer	integer	`X` **or** `Y`
xor	bitwise xor	integer	integer	`X` **xor** `Y`

Table 4.6 Logical (bitwise) operators (continued)

Operator	Operation	Operand types	Result type	Examples
shl	bitwise shift left	integer	integer	`X` **shl** `2`
shr	bitwise shift right	integer	integer	`Y` **shr** `I`

The following rules apply to bitwise operators.

- The result of a **not** operation is of the same type as the operand.
- If the operands of an **and**, **or**, or **xor** operation are both integers, the result is of the predefined integer type with the smallest range that includes all possible values of both types.
- The operations *x* **shl** *y* and *x* **shr** *y* shift the value of *x* to the left or right by *y* bits, which is equivalent to multiplying or dividing *x* by 2^y; the result is of the same type as *x*. For example, if *N* stores the value 01101 (decimal 13), then `N shl 1` returns 11010 (decimal 26). Note that the value of y is interpreted modulo the size of the type of x. Thus for example, if x is an integer, x shl 40 is interpreted as x shl 8 because an integer is 32 bits and 40 mod 32 is 8.

String operators

The relational operators =, <>, <, >, <=, and >= all take string operands (see "Relational operators" on page 4-11). The + operator concatenates two strings.

Table 4.7 String operators

Operator	Operation	Operand types	Result type	Example
+	concatenation	string, packed string, character	string	`S + '. '`

The following rules apply to string concatenation.

- The operands for + can be strings, packed strings (packed arrays of type *Char*), or characters. However, if one operand is of type *WideChar*, the other operand must be a long string.
- The result of a + operation is compatible with any string type. However, if the operands are both short strings or characters, and their combined length is greater than 255, the result is truncated to the first 255 characters.

Pointer operators

The relational operators <, >, <=, and >= can take operands of type *PChar* (see "Relational operators" on page 4-11). The following operators also take pointers as operands. For more information about pointers, see "Pointers and pointer types" on page 5-25.

Table 4.8 Character-pointer operators

Operator	Operation	Operand types	Result type	Example
+	pointer addition	character pointer, integer	character pointer	`P + I`
-	pointer subtraction	character pointer, integer	character pointer, integer	`P - Q`

Table 4.8 Character-pointer operators (continued)

Operator	Operation	Operand types	Result type	Example
^	pointer dereference	pointer	base type of pointer	`P^`
=	equality	pointer	*Boolean*	`P = Q`
<>	inequality	pointer	*Boolean*	`P <> Q`

The **^** operator dereferences a pointer. Its operand can be a pointer of any type except the generic *Pointer*, which must be typecast before dereferencing.

P **=** *Q* is *True* just in case *P* and *Q* point to the same address; otherwise, *P* **<>** *Q* is *True*.

You can use the **+** and **–** operators to increment and decrement the offset of a character pointer. You can also use **–** to calculate the difference between the offsets of two character pointers. The following rules apply.

- If *I* is an integer and *P* is a character pointer, then *P* **+** *I* adds *I* to the address given by *P*; that is, it returns a pointer to the address *I* characters after *P*. (The expression *I* **+** *P* is equivalent to *P* **+** *I*.) *P* **–** *I* subtracts *I* from the address given by *P*; that is, it returns a pointer to the address *I* characters before *P*.
- If *P* and *Q* are both character pointers, then *P* **–** *Q* computes the difference between the address given by *P* (the higher address) and the address given by *Q* (the lower address); that is, it returns an integer denoting the number of characters between *P* and *Q*. *P* **+** *Q* is not defined.

Set operators

The following operators take sets as operands.

Table 4.9 Set operators

Operator	Operation	Operand types	Result type	Example
+	union	set	set	`Set1 + Set2`
–	difference	set	set	`S - T`
*	intersection	set	set	`S * T`
<=	subset	set	*Boolean*	`Q <= MySet`
>=	superset	set	*Boolean*	`S1 >= S2`
=	equality	set	*Boolean*	`S2 = MySet`
<>	inequality	set	*Boolean*	`MySet <> S1`
in	membership	ordinal, set	*Boolean*	`A` **in** `Set1`

The following rules apply to **+**, **–**, and *.

- An ordinal *O* is in *X* **+** *Y* if and only if *O* is in *X* or *Y* (or both). *O* is in *X* **–** *Y* if and only if *O* is in *X* but not in *Y*. *O* is in *X* * *Y* if and only if *O* is in both *X* and *Y*.
- The result of a **+**, **–**, or * operation is of the type **set of** *A*..*B*, where *A* is the smallest ordinal value in the result set and *B* is the largest.

The following rules apply to **<=**, **>=**, **=**, **<>**, and **in**.

- *X* <= *Y* is *True* just in case every member of *X* is a member of *Y*; *Z* >= *W* is equivalent to *W* <= *Z*. *U* = *V* is *True* just in case *U* and *V* contain exactly the same members; otherwise, *U* <> *V* is *True*.
- For an ordinal *O* and a set *S*, *O* **in** *S* is *True* just in case *O* is a member of *S*.

Relational operators

Relational operators are used to compare two operands. The operators =, <>, <=, and >= also apply to sets (see "Set operators" on page 4-10); = and <> also apply to pointers (see "Pointer operators" on page 4-9).

Table 4.10 Relational operators

Operator	Operation	Operand types	Result type	Example
=	equality	simple, class, class reference, interface, string, packed string	*Boolean*	`I = Max`
<>	inequality	simple, class, class reference, interface, string, packed string	*Boolean*	`X <> Y`
<	less-than	simple, string, packed string, *PChar*	*Boolean*	`X < Y`
>	greater-than	simple, string, packed string, *PChar*	*Boolean*	`Len > 0`
<=	less-than-or-equal-to	simple, string, packed string, *PChar*	*Boolean*	`Cnt <= I`
>=	greater-than-or-equal-to	simple, string, packed string, *PChar*	*Boolean*	`I >= 1`

For most simple types, comparison is straightforward. For example, *I* = *J* is *True* just in case *I* and *J* have the same value, and *I* <> *J* is *True* otherwise. The following rules apply to relational operators.

- Operands must be of compatible types, except that a real and an integer can be compared.
- Strings are compared according to the ordering of the extended ASCII character set. Character types are treated as strings of length 1.
- Two packed strings must have the same number of components to be compared. When a packed string with *n* components is compared to a string, the packed string is treated as a string of length *n*.
- The operators <, >, <=, and >= apply to *PChar* operands only if the two pointers point within the same character array.
- The operators = and <> can take operands of class and class-reference types. With operands of a class type, = and <> are evaluated according the rules that apply to pointers: *C* = *D* is *True* just in case *C* and *D* point to the same instance object, and *C* <> *D* is *True* otherwise. With operands of a class-reference type, *C* = *D* is *True* just in case *C* and *D* denote the same class, and *C* <> *D* is *True* otherwise. For more information about classes, see Chapter 7, "Classes and objects".

Class operators

The operators **as** and **is** take classes and instance objects as operands; **as** operates on interfaces as well. For more information, see Chapter 7, "Classes and objects" and Chapter 10, "Object interfaces".

The relational operators **=** and **<>** also operate on classes. See "Relational operators" on page 4-11.

The @ operator

The @ operator returns the address of a variable, or of a function, procedure, or method; that is, @ constructs a pointer to its operand. For more information about pointers, see "Pointers and pointer types" on page 5-25. The following rules apply to @.

- If *X* is a variable, @*X* returns the address of *X*. (Special rules apply when *X* is a procedural variable; see "Procedural types in statements and expressions" on page 5-30.) The type of @*X* is *Pointer* if the default **{$T–}** compiler directive is in effect. In the **{$T+}** state, @*X* is of type **^***T*, where *T* is the type of *X*.
- If *F* is a routine (a function or procedure), @*F* returns *F*'s entry point. The type of @*F* is always *Pointer*.
- When @ is applied to a method defined in a class, the method identifier must be qualified with the class name. For example,

  ```
  @TMyClass.DoSomething
  ```

  points to the *DoSomething* method of *TMyClass*. For more information about classes and methods, see Chapter 7, "Classes and objects".

Operator precedence rules

In complex expressions, rules of precedence determine the order in which operations are performed.

Table 4.11 Precedence of operators

Operators	Precedence
@, **not**	first (highest)
*, /, **div**, **mod**, **and**, **shl**, **shr**, **as**	second
+, –, **or**, **xor**	third
=, <>, <, >, <=, >=, **in**, **is**	fourth (lowest)

An operator with higher precedence is evaluated before an operator with lower precedence, while operators of equal precedence associate to the left. Hence the expression

```
X + Y * Z
```

multiplies *Y* times *Z*, then adds *X* to the result; * is performed first, because is has a higher precedence than **+**. But

```
X - Y + Z
```

first subtracts *Y* from *X*, then adds *Z* to the result; **–** and **+** have the same precedence, so the operation on the left is performed first.

You can use parentheses to override these precedence rules. An expression within parentheses is evaluated first, then treated as a single operand. For example,

```
(X + Y) * Z
```

multiplies *Z* times the sum of *X* and *Y*.

Parentheses are sometimes needed in situations where, at first glance, they seem not to be. For example, consider the expression

```
X = Y or X = Z
```

The intended interpretation of this is obviously

```
(X = Y) or (X = Z)
```

Without parentheses, however, the compiler follows operator precedence rules and reads it as

```
(X = (Y or X)) = Z
```

—which results in a compilation error unless *Z* is Boolean.

Parentheses often make code easier to write and to read, even when they are, strictly speaking, superfluous. Thus the first example above could be written as

```
X + (Y * Z)
```

Here the parentheses are unnecessary (to the compiler), but they spare both programmer and reader from having to think about operator precedence.

Function calls

Because functions return a value, function calls are expressions. For example, if you've defined a function called *Calc* that takes two integer arguments and returns an integer, then the function call `Calc(24, 47)` is an integer expression. If *I* and *J* are integer variables, then `I + Calc(J, 8)` is also an integer expression. Examples of function calls include

```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
GetValue
TSomeObject.SomeMethod(I,J);
```

For more information about functions, see Chapter 6, "Procedures and functions".

Set constructors

A set constructor denotes a set-type value. For example,

```
[5, 6, 7, 8]
```

denotes the set whose members are 5, 6, 7, and 8. The set constructor

```
[ 5..8 ]
```

could also denote the same set.

The syntax for a set constructor is

[*item*$_1$, ..., *item*$_n$]

where each *item* is either an expression denoting an ordinal of the set's base type or a pair of such expressions with two dots (**..**) in between. When an *item* has the form *x***..***y*, it is shorthand for all the ordinals in the range from *x* to *y*, inclusive; but if *x* is greater than *y*, then *x***..***y* denotes nothing and **[***x***..***y***]** is the empty set. The set constructor `[ ]` denotes the empty set, while **[***x***]** denotes the set whose only member is the value of *x*.

Examples of set constructors:

```
[red, green, MyColor]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Digit + 48)]
```

For more information about sets, see "Sets" on page 5-17.

Indexes

Strings, arrays, array properties, and pointers to strings or arrays can be indexed. For example, if *FileName* is a string variable, the expression `FileName[3]` returns the third character in the string denoted by *FileName*, while `FileName[I + 1]` returns the character immediately after the one indexed by *I*. For information about strings, see "String types" on page 5-10. For information about arrays and array properties, see "Arrays" on page 5-18 and "Array properties" on page 7-19.

Typecasts

It is sometimes useful to treat an expression as if it belonged to different type. A typecast allows you to do this by, in effect, temporarily changing an expression's type. For example, `Integer('A')` casts the character *A* as an integer.

The syntax for a typecast is

typeIdentifier(*expression*)

If the expression is a variable, the result is called a *variable typecast*; otherwise, the result is a *value typecast*. While their syntax is the same, different rules apply to the two kinds of typecast.

Value typecasts

In a value typecast, the type identifier and the cast expression must both be ordinal or pointer types. Examples of value typecasts include

```
Integer('A')
Char(48)
Boolean(0)
```

```
Color(2)
Longint(@Buffer)
```

The resulting value is obtained by converting the expression in parentheses. This may involve truncation or extension if the size of the specified type differs from that of the expression. The expression's sign is always preserved.

The statement

```
I := Integer('A');
```

assigns the value of `Integer('A')`—that is, 65—to the variable *I*.

A value typecast cannot be followed by qualifiers and cannot appear on the left side of an assignment statement.

Variable typecasts

You can cast any variable to any type, provided their sizes are the same and you do not mix integers with reals. (To convert numeric types, rely on standard functions like *Int* and *Trunc*.) Examples of variable typecasts include

```
Char(I)
Boolean(Count)
TSomeDefinedType(MyVariable)
```

Variable typecasts can appear on either side of an assignment statement. Thus

```
var MyChar: char;
⋮
Shortint(MyChar) := 122;
```

assigns the character *z* (ASCII 122) to *MyChar*.

You can cast variables to a procedural type. For example, given the declarations

```
type Func = function(X: Integer): Integer;
var
  F: Func;
  P: Pointer;
  N: Integer;
```

you can make the following assignments.

```
F := Func(P);           { Assign procedural value in P to F }
Func(P) := F;           { Assign procedural value in F to P }
@F := P;                { Assign pointer value in P to F }
P := @F;                { Assign pointer value in F to P }
N := F(N);              { Call function via F }
N := Func(P)(N);        { Call function via P }
```

Variable typecasts can also be followed by qualifiers, as illustrated in the following example.

```
type
  TByteRec = record
    Lo, Hi: Byte;
  end;
  TWordRec = record
    Low, High: Word;
    end;
    PByte = ^Byte;
  var
    B: Byte;
    W: Word;
    L: Longint;
    P: Pointer;
  begin
    W := $1234;
    B := TByteRec(W).Lo;
    TByteRec(W).Hi := 0;
    L := $01234567;
    W := TWordRec(L).Low;
    B := TByteRec(TWordRec(L).Low).Hi;
    B := PByte(L)^;
  end;
```

In this example, *TByteRec* is used to access the low- and high-order bytes of a word, and *TWordRec* to access the low- and high-order words of a long integer. You could call the predefined functions *Lo* and *Hi* for the same purpose, but a variable typecast has the advantage that it can be used on the left side of an assignment statement.

For information about typecasting pointers, see "Pointers and pointer types" on page 5-25. For information about casting class and interface types, see "The as operator" on page 7-25 and "Interface typecasts" on page 10-10.

Declarations and statements

Aside from the **uses** clause (and reserved words like **implementation** that demarcate parts of a unit), a program consists entirely of *declarations* and *statements*, which are organized into *blocks*.

Declarations

The names of variables, constants, types, fields, properties, procedures, functions, programs, units, libraries, and packages are called *identifiers*. (Numeric constants like `26057` are not identifiers.) Identifiers must be *declared* before you can use them; the only exceptions are a few predefined types, routines, and constants that the compiler understands automatically, the variable *Result* when it occurs inside a function block, and the variable *Self* when it occurs inside a method implementation.

A declaration defines an identifier and, where appropriate, allocates memory for it. For example,

```
var Size: Extended;
```

declares a variable called *Size* that holds an *Extended* (real) value, while

```
function DoThis(X, Y: string): Integer;
```

declares a function called *DoThis* that takes two strings as arguments and returns an integer. Each declaration ends with a semicolon. When you declare several variables, constants, types, or labels at the same time, you need only write the appropriate reserved word once:

```
var
  Size: Extended;
  Quantity: Integer;
  Description: string;
```

The syntax and placement of a declaration depend on the kind of identifier you are defining. In general, declarations can occur only at the beginning of a block or at the beginning of the interface or implementation section of a unit (after the **uses** clause). Specific conventions for declaring variables, constants, types, functions, and so forth are explained in the chapters on those topics.

The "hint" directives **platform**, **deprecated**, and **library** may be appended to any declaration. In the case of a procedure or function declaration, the hint directive should be separated from the rest of the declaration with a semicolon. Examples:

```
procedure SomeOldRoutine; stdcall; deprecated;

var VersionNumber: Real library;

type AppError = class(Exception)
  ⋮
end platform;
```

When source code is compiled in the **{$HINTS ON} {$WARNINGS ON}** state, each reference to an identifier declared with one of these directives generates an appropriate hint or warning. Use **platform** to mark items that are specific to a particular operating environment (such as Windows or Linux), **deprecated** to indicate that an item is obsolete or supported only for backward compatibility, and **library** to flag dependencies on a particular library or component framework (such as VCL or CLX).

Statements

Statements define algorithmic actions within a program. Simple statements—like assignments and procedure calls—can combine to form loops, conditional statements, and other structured statements.

Multiple statements within a block, and in the initialization or finalization section of a unit, are separated by semicolons.

Simple statements

A simple statement doesn't contain any other statements. Simple statements include assignments, calls to procedures and functions, and **goto** jumps.

Assignment statements

An assignment statement has the form

variable `:=` *expression*

where *variable* is any variable reference—including a variable, variable typecast, dereferenced pointer, or component of a structured variable—and *expression* is any assignment-compatible expression. (Within a function block, *variable* can be replaced with the name of the function being defined. See Chapter 6, "Procedures and functions".) The **:=** symbol is sometimes called the *assignment operator*.

An assignment statement replaces the current value of *variable* with the value of *expression*. For example,

```
I := 3;
```

assigns the value 3 to the variable *I*. The variable reference on the left side of the assignment can appear in the expression on the right. For example,

```
I := I + 1;
```

increments the value of *I*. Other assignment statements include

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I  * K;
Shortint(MyChar) := 122;
TByteRec(W).Hi := 0;
MyString[I] := 'A';
SomeArray[I + 1] := P^;
TMyObject.SomeProperty := True;
```

Procedure and function calls

A procedure call consists of the name of a procedure (with or without qualifiers), followed by a parameter list (if required). Examples include

```
PrintHeading;
Transpose(A, N, M);
Find(Smith, William);
Writeln('Hello world!');
DoSomething();
Unit1.SomeProcedure;
TMyObject.SomeMethod(X,Y);
```

With extended syntax enabled (**{$X+}**), function calls, like calls to procedures, can be treated as statements in their own right:

```
MyFunction(X);
```

When you use a function call in this way, its return value is discarded.

For more information about procedures and functions, see Chapter 6, "Procedures and functions".

Goto statements

A **goto** statement, which has the form

```
goto label
```

transfers program execution to the statement marked by the specified label. To mark a statement, you must first declare the label. Then precede the statement you want to mark with the label and a colon:

label: *statement*

Declare labels like this:

`label` *label*;

You can declare several labels at once:

`label` *label*$_1$, ..., *label*$_n$;

A label can be any valid identifier or any numeral between 0 and 9999.

The label declaration, marked statement, and **goto** statement must belong to the same block. (See "Blocks and scope" on page 4-27.) Hence it is not possible to jump into or out of a procedure or function. Do not mark more than one statement in a block with the same label.

For example,

```
label StartHere;
⋮
StartHere: Beep;
goto StartHere;
```

creates an infinite loop that calls the *Beep* procedure repeatedly.

The **goto** statement is generally discouraged in structured programming. It is, however, sometimes used as a way of exiting from nested loops, as in the following example.

```
procedure FindFirstAnswer;
var X, Y, Z, Count: Integer;
label FoundAnAnswer;
begin
  Count := SomeConstant;
  for X := 1 to Count do
    for Y := 1 to Count do
      for Z := 1 to Count do
        if ... { some condition holds on X, Y, and Z } then
          goto FoundAnAnswer;

  ⋮ {code to execute if no answer is found }
  Exit;

  FoundAnAnswer:
    ⋮ { code to execute when an answer is found }
end;
```

Notice that we are using **goto** to jump *out* of a nested loop. Never jump *into* a loop or other structured statement, since this can have unpredictable effects.

Structured statements

Structured statements are built from other statements. Use a structured statement when you want to execute other statements sequentially, conditionally, or repeatedly.

- A compound or **with** statement simply executes a sequence of constituent statements.
- A conditional statement—that is, an **if** or **case** statement—executes at most one of its constituents, depending on specified criteria.
- Loop statements—including **repeat**, **while**, and **for** loops—execute a sequence of constituent statements repeatedly.
- A special group of statements—including **raise**, **try...except**, and **try...finally** constructions—create and handle *exceptions*. For information about exception generation and handling, see "Exceptions" on page 7-26.

Compound statements

A compound statement is a sequence of other (simple or structured) statements to be executed in the order in which they are written. The compound statement is bracketed by the reserved words **begin** and **end**, and its constituent statements are separated by semicolons. For example:

```
begin
  Z := X;
  X := Y;
  Y := Z;
end;
```

The last semicolon before **end** is optional. So we could have written this as

```
begin
  Z := X;
  X := Y;
  Y := Z
end;
```

Compound statements are essential in contexts where Object Pascal syntax requires a single statement. In addition to program, function, and procedure blocks, they occur within other structured statements, such as conditionals or loops. For example:

```
begin
  I := SomeConstant;
  while I > 0 do
  begin
    ⋮
    I := I - 1;
  end;
end;
```

You can write a compound statement that contains only a single constituent statement; like parentheses in a complex term, **begin** and **end** sometimes serve to disambiguate and to improve readability. You can also use an empty compound statement to create a block that does nothing:

```
begin
end;
```

With statements

A **with** statement is a shorthand for referencing the fields of a record or the fields, properties, and methods of an object. The syntax of a **with** statement is

`with` *obj* `do` *statement*

or

`with` obj_1, ..., obj_n `do` *statement*

where *obj* is a variable reference denoting an object or record, and *statement* is any simple or structured statement. Within *statement*, you can refer to fields, properties, and methods of *obj* using their identifiers alone—without qualifiers.

For example, given the declarations

```
type TDate = record
  Day: Integer;
  Month: Integer;
  Year: Integer;
end;

var OrderDate: TDate;
```

you could write the following **with** statement.

```
with OrderDate do
  if Month = 12 then
  begin
    Month := 1;
    Year := Year + 1;
  end
  else
    Month := Month + 1;
```

This is equivalent to

```
if OrderDate.Month = 12 then
begin
  OrderDate.Month := 1;
  OrderDate.Year := OrderDate.Year + 1;
end
else
  OrderDate.Month := OrderDate.Month + 1;
```

If the interpretation of *obj* involves indexing arrays or dereferencing pointers, these actions are performed once, before *statement* is executed. This makes **with** statements efficient as well as concise. It also means that assignments to a variable within *statement* cannot affect the interpretation of *obj* during the current execution of the **with** statement.

Each variable reference or method name in a **with** statement is interpreted, if possible, as a member of the specified object or record. If there is another variable or method of the same name that you want to access from the **with** statement, you need to prepend it with a qualifier, as in the following example.

```
with OrderDate do
  begin
    Year := Unit1.Year
    ⋮
  end;
```

When multiple objects or records appear after **with**, the entire statement is treated like a series of nested **with** statements. Thus

```
with obj1, obj2, ..., objn do statement
```

is equivalent to

```
with obj1 do
  with obj2 do
     ⋮
     with objn do
       statement
```

In this case, each variable reference or method name in *statement* is interpreted, if possible, as a member of obj_n; otherwise it is interpreted, if possible, as a member of obj_{n-1}; and so forth. The same rule applies to interpreting the *obj*s themselves, so that, for instance, if obj_n is a member of both obj_1 and obj_2, it is interpreted as $obj_2.obj_n$.

If statements

There are two forms of **if** statement: **if...then** and the **if...then...else**. The syntax of an **if...then** statement is

```
if expression then statement
```

where *expression* returns a Boolean value. If *expression* is *True*, then *statement* is executed; otherwise it is not. For example,

```
if J <> 0 then Result := I/J;
```

The syntax of an **if...then...else** statement is

```
if expression then statement1 else statement2
```

where *expression* returns a Boolean value. If *expression* is *True*, then $statement_1$ is executed; otherwise $statement_2$ is executed. For example,

```
if J = 0 then
  Exit
else
  Result := I/J;
```

The **then** and **else** clauses contain one statement each, but it can be a structured statement. For example,

```
if J <> 0 then
begin
  Result := I/J;
  Count := Count + 1;
end
```

```
else if Count = Last then
  Done := True
else
  Exit;
```

Notice that there is never a semicolon between the **then** clause and the word **else**. You can place a semicolon after an entire **if** statement to separate it from the next statement in its block, but the **then** and **else** clauses require nothing more than a space or carriage return between them. Placing a semicolon immediately before **else** (in an **if** statement) is a common programming error.

A special difficulty arises in connection with nested **if** statements. The problem arises because some **if** statements have **else** clauses while others do not, but the syntax for the two kinds of statement is otherwise the same. In a series of nested conditionals where there are fewer **else** clauses than **if** statements, it may not seem clear which **else** clauses are bound to which **if**s. Consider a statement of the form

`if` $\mathit{expression}_1$ `then if` $\mathit{expression}_2$ `then` $\mathit{statement}_1$ `else` $\mathit{statement}_2$`;`

There would appear to be two ways to parse this:

`if` $\mathit{expression}_1$ `then` [`if` $\mathit{expression}_2$ `then` $\mathit{statement}_1$ `else` $\mathit{statement}_2$]`;`

`if` $\mathit{expression}_1$ `then` [`if` $\mathit{expression}_2$ `then` $\mathit{statement}_1$] `else` $\mathit{statement}_2$`;`

The compiler always parses in the first way. That is, in real code, the statement

```
if ... { expression1 } then
  if ... { expression2 } then
    ... { statement1 }
  else
    ... { statement2 } ;
```

is equivalent to

```
if ... { expression1 } then
begin
  if ... { expression2 } then
    ... { statement1 }
  else
    ... { statement2 }
end;
```

The rule is that nested conditionals are parsed starting from the innermost conditional, with each **else** bound to the nearest available **if** on its left. To force the compiler to read our example in the second way, you would have to write it explicitly as

```
if ... { expression1 } then
begin
  if ... { expression2 } then
    ... { statement1 }
end
else
  ... { statement2 } ;
```

Case statements

The **case** statement provides a readable alternative to complex nested **if** conditionals. A **case** statement has the form

`case` *selectorExpression* `of`
caseList$_1$`:` *statement*$_1$`;`
⋮
caseList$_n$`:` *statement*$_n$`;`
`end`

where *selectorExpression* is any expression of an ordinal type (string types are invalid) and each *caseList* is one of the following:

- A numeral, declared constant, or other expression that the compiler can evaluate without executing your program. It must be of an ordinal type compatible with *selectorExpression*. Thus `7`, `True`, `4 + 5 * 3`, `'A'`, and `Integer('A')` can all be used as *caseList*s, but variables and most function calls cannot. (A few built-in functions like *Hi* and *Lo* can occur in a *caseList*. See “Constant expressions” on page 5-41.)
- A subrange having the form *First..Last*, where *First* and *Last* both satisfy the criterion above and *First* is less than or equal to *Last*.
- A list having the form *item*$_1$`,` ..., *item*$_n$, where each *item* satisfies one of the criteria above.

Each value represented by a *caseList* must be unique in the **case** statement; subranges and lists cannot overlap. A **case** statement can have a final **else** clause:

`case` *selectorExpression* `of`
caseList$_1$`:` *statement*$_1$`;`
⋮
caseList$_n$`:` *statement*$_n$`;`
`else`
statements`;`
`end`

where *statements* is a semicolon-delimited sequence of statements. When a **case** statement is executed, at most one of *statement*$_1$ ... *statement*$_n$ is executed. Whichever *caseList* has a value equal to that of *selectorExpression* determines the *statement* to be used. If none of the *caseList*s has the same value as *selectorExpression*, then the statements in the **else** clause (if there is one) are executed.

The **case** statement

```
case I of
  1..5: Caption := 'Low';
  6..9: Caption := 'High';
  0, 10..99: Caption := 'Out of range';
else
  Caption := '';
end;
```

is equivalent to the nested conditional

```
if I in [1..5] then
  Caption := 'Low'
```

```
    else if I in [6..10] then
      Caption := 'High'
      else if (I = 0) or (I in [10..99]) then
        Caption := 'Out of range'
        else
          Caption := '';
```

Other examples of **case** statements:

```
case MyColor of
  Red: X := 1;
  Green: X := 2;
  Blue: X := 3;
  Yellow, Orange, Black: X := 0;
end;

case Selection of
  Done: Form1.Close;
  Compute: CalculateTotal(UnitCost, Quantity);
else
  Beep;
end;
```

Control loops

Loops allow you to execute a sequence of statements repeatedly, using a control condition or variable to determine when the execution stops. Object Pascal has three kinds of control loop: **repeat** statements, **while** statements, and **for** statements.

You can use the standard *Break* and *Continue* procedures to control the flow of a **repeat**, **while**, or **for** statement. *Break* terminates the statement in which it occurs, while *Continue* begins executing the next iteration of the sequence. For more information about these procedures, see the online Help.

Repeat statements

The syntax of a **repeat** statement is

`repeat` *statement*$_1$`;` ...`;` *statement*$_n$`;` `until` *expression*

where *expression* returns a Boolean value. (The last semicolon before **until** is optional.) The **repeat** statement executes its sequence of constituent statements continually, testing *expression* after each iteration. When *expression* returns *True*, the **repeat** statement terminates. The sequence is always executed at least once because *expression* is not evaluated until after the first iteration.

Examples of **repeat** statements include

```
repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;

repeat
  Write('Enter a value (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);
```

While statements

A **while** statement is similar to a **repeat** statement, except that the control condition is evaluated before the first execution of the statement sequence. Hence, if the condition is false, the statement sequence is never executed.

The syntax of a **while** statement is

`while` *expression* `do` *statement*

where *expression* returns a Boolean value and *statement* can be a compound statement. The **while** statement executes its constituent *statement* repeatedly, testing *expression* before each iteration. As long as *expression* returns *True*, execution continues.

Examples of **while** statements include

```
while Data[I] <> X do I := I + 1;

while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;

while not Eof(InputFile) do
begin
  Readln(InputFile, Line);
  Process(Line);
end;
```

For statements

A **for** statement, unlike a **repeat** or **while** statement, requires you to specify explicitly the number of iterations you want the loop to go through. The syntax of a **for** statement is

`for` *counter* `:=` *initialValue* `to` *finalValue* `do` *statement*

or

`for` *counter* `:=` *initialValue* `downto` *finalValue* `do` *statement*

where

- *counter* is a local variable (declared in the block containing the **for** statement) of ordinal type, without any qualifiers.
- *initialValue* and *finalValue* are expressions that are assignment-compatible with *counter*.
- *statement* is a simple or structured statement that does not change the value of *counter*.

The **for** statement assigns the value of *initialValue* to *counter*, then executes *statement* repeatedly, incrementing or decrementing *counter* after each iteration. (The **for...to** syntax increments *counter*, while the **for...downto** syntax decrements it.) When *counter* returns the same value as *finalValue*, *statement* is executed once more and the **for** statement terminates. In other words, *statement* is executed once for every value in the range from *initialValue* to *finalValue*. If *initialValue* is equal to *finalValue*, *statement* is executed exactly once. If *initialValue* is greater than *finalValue* in a **for...to** statement, or less than *finalValue* in a **for...downto** statement, then *statement* is never executed. After the **for** statement terminates, the value of *counter* is undefined.

For purposes of controlling execution of the loop, the expressions *initialValue* and *finalValue* are evaluated only once, before the loop begins. Hence the **for...to** statement is almost, but not quite, equivalent to this **while** construction:

```
begin
  counter := initialValue;
  while counter <= finalValue do
  begin
    statement;
    counter := Succ(counter);
  end;
end
```

The difference between this construction and the **for...to** statement is that the **while** loop reevaluates *finalValue* before each iteration. This can result in noticeably slower performance if *finalValue* is a complex expression, and it also means that changes to the value of *finalValue* within *statement* can affect execution of the loop.

Examples of **for** statements:

```
for I := 2 to 63 do
  if Data[I] > Max then
    Max := Data[I];

for I := ListBox1.Items.Count - 1 downto 0 do
  ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);

for I := 1 to 10 do
  for J := 1 to 10 do
  begin
    X := 0;
    for K := 1 to 10 do
      X := X + Mat1[I, K] * Mat2[K, J];
    Mat[I, J] := X;
  end;

for C := Red to Blue do Check(C);
```

Blocks and scope

Declarations and statements are organized into *blocks*, which define local namespaces (or *scopes*) for labels and identifiers. Blocks allow a single identifier, such as a variable name, to have different meanings in different parts of a program. Each block is part of the declaration of a program, function, or procedure; each program, function, or procedure declaration has one block.

Blocks

A block consists of a series of declarations followed by a compound statement. All declarations must occur together at the beginning of the block. So the form of a block is

```
declarations
begin
  statements
end
```

The *declarations* section can include, in any order, declarations for variables, constants (including resource strings), types, procedures, functions, and labels. In a program block, the *declarations* section can also include one or more **exports** clauses (see Chapter 9, "Libraries and packages").

For example, in a function declaration like

```
function UpperCase(const S: string): string;
var
  Ch: Char;
  L: Integer;
  Source, Dest: PChar;
begin
  ⋮
end;
```

the first line of the declaration is the function heading and all of the succeeding lines make up the block. *Ch*, *L*, *Source*, and *Dest* are local variables; their declarations apply only to the *UpperCase* function block and override—in this block only—any declarations of the same identifiers that may occur in the program block or in the interface or implementation section of a unit.

Scope

An identifier, such as a variable or function name, can be used only within the *scope* of its declaration. The location of a declaration determines its scope. An identifier declared within the declaration of a program, function, or procedure has a scope limited to the block in which it is declared. An identifier declared in the interface section of a unit has a scope that includes any other units or programs that use the unit where the declaration occurs. Identifiers with narrower scope—especially identifiers declared in functions and procedures—are sometimes called *local*, while identifiers with wider scope are called *global*.

The rules that determine identifier scope are summarized below.

If the identifier is declared in ...	its scope extends ...
the declaration of a program, function, or procedure	from the point where it is declared to the end of the current block, including all blocks enclosed within that scope.
the interface section of a unit	from the point where it is declared to the end of the unit, and to any other unit or program that uses that unit. (See Chapter 3, "Programs and units".)
the implementation section of a unit, but not within the block of any function or procedure	from the point where it is declared to the end of the unit. The identifier is available to any function or procedure in the unit, including the initialization and finalization sections, if present.
the definition of a record type (that is, the identifier is the name of a field in the record)	from the point of its declaration to the end of the record-type definition. (See "Records" on page 5-21.)
the definition of a class (that is, the identifier is the name of a data field property or method in the class)	from the point of its declaration to the end of the class-type definition, and also includes descendants of the class and the blocks of all methods in the class and its descendants. (See Chapter 7, "Classes and objects".)

Naming conflicts

When one block encloses another, the former is called the *outer block* and the latter the *inner block*. If an identifier declared in an outer block is redeclared in an inner block, the inner declaration overrides the outer one and determines the meaning of the identifier for the duration of the inner block. For example, if you declare a variable called *MaxValue* in the interface section of a unit, and then declare another variable with the same name in a function declaration within that unit, any unqualified occurrences of *MaxValue* in the function block are governed by the second, local declaration. Similarly, a function declared within another function creates a new, inner scope in which identifiers used by the outer function can be redeclared locally.

The use of multiple units further complicates the definition of scope. Each unit listed in a **uses** clause imposes a new scope that encloses the remaining units used and the program or unit containing the **uses** clause. The first unit in a **uses** clause represents the outermost scope and each succeeding unit represents a new scope inside the previous one. If two or more units declare the same identifier in their interface sections, an unqualified reference to the identifier selects the declaration in the innermost scope—that is, in the unit where the reference itself occurs, or, if that unit doesn't declare the identifier, in the last unit in the **uses** clause that does declare the identifier.

The *System* unit is used automatically by every program or unit. The declarations in *System*, along with the predefined types, routines, and constants that the compiler understands automatically, always have the outermost scope.

You can override these rules of scope and bypass an inner declaration by using a qualified identifier (see "Qualified identifiers" on page 4-2) or a **with** statement (see "With statements" on page 4-21).

Chapter

5

Data types, variables, and constants

A *type* is essentially a name for a kind of data. When you declare a variable you must specify its type, which determines the set of values the variable can hold and the operations that can be performed on it. Every expression returns data of a particular type, as does every function. Most functions and procedures require parameters of specific types.

Object Pascal is a “strongly typed” language, which means that it distinguishes a variety of data types and does not always allow you to substitute one type for another. This is usually beneficial because it lets the compiler treat data intelligently and validate your code more thoroughly, preventing hard-to-diagnose runtime errors. When you need greater flexibility, however, there are mechanisms to circumvent strong typing. These include *typecasting* (see “Typecasts” on page 4-14), *pointers* (see “Pointers and pointer types” on page 5-25), *variants* (see “Variant types” on page 5-31), *variant parts* in records (see “Variant parts in records” on page 5-23), and *absolute addressing* of variables (see “Absolute addresses” on page 5-39).

About types

There are several ways to categorize Object Pascal data types:

- Some types are *predefined* (or *built-in*); the compiler recognizes these automatically, without the need for a declaration. Almost all of the types documented in this language reference are predefined. Other types are created by declaration; these include user-defined types and the types defined in the product libraries.
- Types can be classified as either *fundamental* or *generic*. The range and format of a fundamental type is the same in all implementations of Object Pascal, regardless of the underlying CPU and operating system. The range and format of a generic type is platform-specific and could vary across different implementations. Most predefined types are fundamental, but a handful of integer, character, string, and pointer types are generic. It’s a good idea to use generic types when possible, since they provide optimal performance and portability. However, changes in storage

format from one implementation of a generic type to the next could cause compatibility problems—for example, if you are streaming data to a file.

- Types can be classified as *simple*, *string*, *structured*, *pointer*, *procedural*, or *variant*. In addition, type identifiers themselves can be regarded as belonging to a special "type" because they can be passed as parameters to certain functions (such as *High*, *Low*, and *SizeOf*).

The outline below shows the taxonomy of Object Pascal data types.

- **simple**
  - ordinal
    - integer
    - character
    - Boolean
    - enumerated
    - subrange
  - real
- **string**
- **structured**
  - set
  - array
  - record
  - file
  - class
  - class reference
  - interface
- **pointer**
- **procedural**
- **variant**
- **type identifier**

The standard function *SizeOf* operates on all variables and type identifiers. It returns an integer representing the amount of memory (in bytes) used to store data of the specified type. For example, `SizeOf(Longint)` returns 4, since a *Longint* variable uses four bytes of memory.

Type declarations are illustrated in the sections that follow. For general information about type declarations, see "Declaring types" on page 5-37.

Simple types

Simple types, which include *ordinal* types and *real* types, define ordered sets of values.

Ordinal types

Ordinal types include *integer*, *character*, *Boolean*, *enumerated*, and *subrange* types. An ordinal type defines an ordered set of values in which each value except the first has a unique *predecessor* and each value except the last has a unique *successor*. Further, each value has an *ordinality* which determines the ordering of the type. In most cases, if a value has ordinality n, its predecessor has ordinality $n-1$ and its successor has ordinality $n+1$.

- For integer types, the ordinality of a value is the value itself.
- Subrange types maintain the ordinalities of their base types.
- For other ordinal types, by default the first value has ordinality 0, the next value has ordinality 1, and so forth. The declaration of an enumerated type can explicitly override this default.

Several predefined functions operate on ordinal values and type identifiers. The most important of them are summarized below.

Function	Parameter	Return value	Remarks
Ord	ordinal expression	ordinality of expression's value	Does not take *Int64* arguments.
Pred	ordinal expression	predecessor of expression's value	
Succ	ordinal expression	successor of expression's value	
High	ordinal type identifier or variable of ordinal type	highest value in type	Also operates on short-string types and arrays.
Low	ordinal type identifier or variable of ordinal type	lowest value in type	Also operates on short-string types and arrays.

For example, `High(Byte)` returns 255 because the highest value of type *Byte* is 255, and `Succ(2)` returns 3 because 3 is the successor of 2.

The standard procedures *Inc* and *Dec* increment and decrement the value of an ordinal variable. For example, `Inc(I)` is equivalent to `I := Succ(I)` and, if *I* is an integer variable, to `I := I + 1`.

Integer types

An integer type represents a subset of the whole numbers. The generic integer types are *Integer* and *Cardinal*; use these whenever possible, since they result in the best performance for the underlying CPU and operating system. The table below gives their ranges and storage formats for the current 32-bit Object Pascal compiler.

Table 5.1 Generic integer types for 32-bit implementations of Object Pascal

Type	Range	Format
Integer	–2147483648..2147483647	signed 32-bit
Cardinal	0..4294967295	unsigned 32-bit

Fundamental integer types include *Shortint*, *Smallint*, *Longint*, *Int64*, *Byte*, *Word*, and *Longword*.

Table 5.2 Fundamental integer types

Type	Range	Format
Shortint	–128..127	signed 8-bit
Smallint	–32768..32767	signed 16-bit
Longint	–2147483648..2147483647	signed 32-bit
Int64	$-2^{63}..2^{63}-1$	signed 64-bit
Byte	0..255	unsigned 8-bit
Word	0..65535	unsigned 16-bit
Longword	0..4294967295	unsigned 32-bit

In general, arithmetic operations on integers return a value of type *Integer*—which, in its current implementation, is equivalent to the 32-bit *Longint*. Operations return a value of type *Int64* only when performed on an *Int64* operand. Hence the following code produces incorrect results.

```
var
  I: Integer;
  J: Int64;
  ⋮
I := High(Integer);
J := I + 1;
```

To get an *Int64* return value in this situation, cast *I* as *Int64*:

```
 ⋮
J := Int64(I) + 1;
```

For more information, see “Arithmetic operators” on page 4-6.

Note Most standard routines that take integer arguments truncate *Int64* values to 32 bits. However, the *High*, *Low*, *Succ*, *Pred*, *Inc*, *Dec*, *IntToStr*, and *IntToHex* routines fully support *Int64* arguments. Also, the *Round*, *Trunc*, *StrToInt64*, and *StrToInt64Def* functions return *Int64* values. A few routines—including *Ord*—cannot take *Int64* values at all.

When you increment the last value or decrement the first value of an integer type, the result wraps around the beginning or end of the range. For example, the *Shortint* type has the range –128..127; hence, after execution of the code

```
var I: Shortint;
 ⋮
I := High(Shortint);
I := I + 1;
```

the value of *I* is –128. If compiler range-checking is enabled, however, this code generates a runtime error.

Character types

The fundamental character types are *AnsiChar* and *WideChar*. *AnsiChar* values are byte-sized (8-bit) characters ordered according to the locale character set which is possibly multibyte. *AnsiChar* was originally modeled after the ANSI character set (thus its name) but has now been broadened to refer to the current locale character set.

WideChar characters use more than one byte to represent every character. In the current implementations, *WideChar* is word-sized (16-bit) characters ordered according to the Unicode character set (note that it could be longer in future implementations). The first 256 Unicode characters correspond to the ANSI characters.

Note On Linux, wchar_t widechar is 32 bits per character. The 16-bit Unicode standard that Object Pascal *WideChar*s support is a subset of the 32-bit UCS standard supported by Linux and the GNU libraries. Pascal *WideChar* data must be widened to 32 bits per character before it can be passed to an OS function as wchar_t.

The generic character type is *Char*, which is equivalent to *AnsiChar*. Because the implementation of *Char* is subject to change, it's a good idea to use the standard function *SizeOf* rather than a hard-coded constant when writing programs that may need to handle characters of different sizes.

A string constant of length 1, such as `'A'`, can denote a character value. The predefined function *Chr* returns the character value for any integer in the range of *AnsiChar* or *WideChar*; for example, `Chr(65)` returns the letter *A*.

Character values, like integers, wrap around when decremented or incremented past the beginning or end of their range (unless range-checking is enabled). For example, after execution of the code

```
var
  Letter: Char;
  I: Integer;
begin
  Letter := High(Letter);
  for I := 1 to 66 do
    Inc(Letter);
end;
```

Letter has the value *A* (ASCII 65).

For more information about Unicode characters, see “About extended character sets” on page 5-13 and “Working with null-terminated strings” on page 5-13.

Boolean types

The four predefined Boolean types are *Boolean*, *ByteBool*, *WordBool*, and *LongBool*. *Boolean* is the preferred type. The others exist to provide compatibility with other languages and operating system libraries.

A *Boolean* variable occupies one byte of memory, a *ByteBool* variable also occupies one byte, a *WordBool* variable occupies two bytes (one word), and a *LongBool* variable occupies four bytes (two words).

Boolean values are denoted by the predefined constants *True* and *False*. The following relationships hold.

Boolean	**ByteBool, WordBool, LongBool**
False < *True*	*False* <> *True*
Ord(*False*) = 0	*Ord*(*False*) = 0
Ord(*True*) = 1	*Ord*(*True*) <> 0
Succ(*False*) = *True*	*Succ*(*False*) = *True*
Pred(*True*) = *False*	*Pred*(*False*) = *True*

A value of type *ByteBool*, *LongBool*, or *WordBool* is considered *True* when its ordinality is nonzero. If such a value appears in a context where a *Boolean* is expected, the compiler automatically converts any value of nonzero ordinality to *True*.

The remarks above refer to the ordinality of Boolean values— not to the values themselves. In Object Pascal, Boolean expressions cannot be equated with integers or reals. Hence, if *X* is an integer variable, the statement

```
if X then ...;
```

generates a compilation error. Casting the variable to a Boolean type is unreliable, but each of the following alternatives will work.

```
if X <> 0 then ...;       { use longer expression that returns Boolean value }

var OK: Boolean           { use Boolean variable }
 ⋮
if X <> 0 then OK := True;
if OK then ...;
```

Enumerated types

An enumerated type defines an ordered set of values by simply listing identifiers that denote these values. The values have no inherent meaning. To declare an enumerated type, use the syntax

type *typeName* = (*val*$_1$, ..., *val*$_n$)

where *typeName* and each *val* are valid identifiers. For example, the declaration

```
type Suit = (Club, Diamond, Heart, Spade);
```

defines an enumerated type called *Suit* whose possible values are *Club*, *Diamond*, *Heart*, and *Spade*, where `Ord(Club)` returns 0, `Ord(Diamond)` returns 1, and so forth.

When you declare an enumerated type, you are declaring each *val* to be a constant of type *typeName*. If the *val* identifiers are used for another purpose within the same scope, naming conflicts occur. For example, suppose you declare the type

```
type TSound = (Click, Clack, Clock);
```

Unfortunately, *Click* is also the name of a method defined for *TControl* and all of the objects in the VCL and/or CLX that descend from it. So if you're writing an application and you create an event handler like

```
procedure TForm1.DBGrid1Enter(Sender: TObject);
var Thing: TSound;
begin
  ⋮
  Thing := Click;
  ⋮
end;
```

you'll get a compilation error; the compiler interprets *Click* within the scope of the procedure as a reference to *TForm*'s *Click* method. You can work around this by qualifying the identifier; thus, if *TSound* is declared in *MyUnit*, you would use

```
Thing := MyUnit.Click;
```

A better solution, however, is to choose constant names that are not likely to conflict with other identifiers. Examples:

```
type
  TSound = (tsClick, tsClack, tsClock);
  TMyColor = (mcRed, mcBlue, mcGreen, mcYellow, mcOrange);
  Answer = (ansYes, ansNo, ansMaybe);
```

You can use the `(`*val*$_1$`, ..., `*val*$_n$`)` construction directly in variable declarations, as if it were a type name:

```
var MyCard: (Club, Diamond, Heart, Spade);
```

But if you declare *MyCard* this way, you can't declare another variable within the same scope using these constant identifiers. Thus

```
var Card1: (Club, Diamond, Heart, Spade);
var Card2: (Club, Diamond, Heart, Spade);
```

generates a compilation error. But

```
var Card1, Card2: (Club, Diamond, Heart, Spade);
```

compiles cleanly, as does

```
type Suit = (Club, Diamond, Heart, Spade);
var
  Card1: Suit;
  Card2: Suit;
```

Enumerated types with explicitly assigned ordinality

By default, the ordinalities of enumerated values start from 0 and follow the sequence in which their identifiers are listed in the type declaration. You can override this by explicitly assigning ordinalities to some or all of the values in the declaration. To assign an ordinality to a value, follow its identifier with `=` *constantExpression*, where *constantExpression* is a constant expression that evaluates to an integer. (See "Constant expressions" on page 5-41) For example,

```
type Size = (Small = 5, Medium = 10, Large = Small + Medium);
```

defines a type called *Size* whose possible values include *Small*, *Medium*, and *Large*, where `Ord(Small)` returns 5, `Ord(Medium)` returns 10, and `Ord(Large)` returns 15.

An enumerated type is, in effect, a subrange whose lowest and highest values correspond to the lowest and highest ordinalities of the constants in the declaration.

In the example above, the *Size* type has 11 possible values whose ordinalities range from 5 to 15. (Hence the type `array[Size] of Char` represents an array of 11 characters.) Only three of these values have names, but the others are accessible through typecasts and through routines such as *Pred*, *Succ*, *Inc*, and *Dec*. In the following example, "anonymous" values in the range of *Size* are assigned to the variable *X*.

```
var X: Size;
X := Small;    // Ord(X) = 5
X := Size(6);  // Ord(X) = 6
Inc(X);        // Ord(X) = 7
```

Any value that isn't explicitly assigned an ordinality has ordinality one greater than that of the previous value in the list. If the first value isn't assigned an ordinality, its ordinality is 0. Hence, given the declaration

```
type SomeEnum = (e1, e2, e3 = 1);
```

SomeEnum has only two possible values: `Ord(e1)` returns 0, `Ord(e2)` returns 1, and `Ord(e3)` also returns 1; because *e2* and *e3* have the same ordinality, they represent the same value.

Subrange types

A subrange type represents a subset of the values in another ordinal type (called the *base type*). Any construction of the form *Low..High*, where *Low* and *High* are constant expressions of the same ordinal type and *Low* is less than *High*, identifies a subrange type that includes all values between *Low* and *High*. For example, if you declare the enumerated type

```
type TColors = (Red, Blue, Green, Yellow, Orange, Purple, White, Black);
```

you can then define a subrange type like

```
type TMyColors = Green..White;
```

Here *TMyColors* includes the values *Green*, *Yellow*, *Orange*, *Purple*, and *White*.

You can use numeric constants and characters (string constants of length 1) to define subrange types:

```
type
  SomeNumbers = -128..127;
  Caps = 'A'..'Z';
```

When you use numeric or character constants to define a subrange, the base type is the smallest integer or character type that contains the specified range.

The *Low..High* construction itself functions as a type name, so you can use it directly in variable declarations. For example,

```
var SomeNum: 1..500;
```

declares an integer variable whose value can be anywhere in the range from 1 to 500.

The ordinality of each value in a subrange is preserved from the base type. (In the first example above, if *Color* is a variable that holds the value *Green*, `Ord(Color)` returns 2 regardless of whether *Color* is of type *TColors* or *TMyColors*.) Values do not wrap around the beginning or end of a subrange, even if the base is an integer or character type; incrementing or decrementing past the boundary of a subrange simply converts the value to the base type. Hence, while

```
type Percentile = 0..99;
var I: Percentile;
⋮
I := 100;
```

produces an error,

```
⋮
I := 99;
Inc(I);
```

assigns the value 100 to *I* (unless compiler range-checking is enabled).

The use of constant expressions in subrange definitions introduces a syntactic difficulty. In any type declaration, when the first meaningful character after = is a left parenthesis, the compiler assumes that an enumerated type is being defined. Hence the code

```
const
  X = 50;
  Y = 10;
type
  Scale = (X - Y) * 2..(X + Y) * 2;
```

produces an error. Work around this problem by rewriting the type declaration to avoid the leading parenthesis:

```
type
  Scale = 2 * (X - Y)..(X + Y) * 2;
```

Real types

A real type defines a set of numbers that can be represented with floating-point notation. The table below gives the ranges and storage formats for the fundamental real types.

Table 5.3 Fundamental real types

Type	Range	Significant digits	Size in bytes
Real48	2.9×10^{-39} .. 1.7×10^{38}	11–12	6
Single	1.5×10^{-45} .. 3.4×10^{38}	7–8	4
Double	5.0×10^{-324} .. 1.7×10^{308}	15–16	8
Extended	3.6×10^{-4951} .. 1.1×10^{4932}	19–20	10
Comp	$-2^{63}+1$ .. $2^{63}-1$	19–20	8
Currency	–922337203685477.5808.. 922337203685477.5807	19–20	8

The generic type *Real*, in its current implementation, is equivalent to *Double*.

Table 5.4 Generic real types

Type	Range	Significant digits	Size in bytes
Real	5.0×10^{-324} .. 1.7×10^{308}	15–16	8

Note The six-byte *Real48* type was called *Real* in earlier versions of Object Pascal. If you are recompiling code that uses the older, six-byte *Real* type, you may want to change it to *Real48*. You can also use the **{$REALCOMPATIBILITY ON}** compiler directive to turn *Real* back into the six-byte type.

The following remarks apply to fundamental real types.

- *Real48* is maintained for backward compatibility. Since its storage format is not native to the Intel CPU family, it results in slower performance than other floating-point types.
- *Extended* offers greater precision than other real types but is less portable. Be careful using *Extended* if you are creating data files to share across platforms.
- The *Comp* (computational) type is native to the Intel CPU and represents a 64-bit integer. It is classified as a real, however, because it does not behave like an ordinal type. (For example, you cannot increment or decrement a *Comp* value.) *Comp* is maintained for backward compatibility only. Use the *Int64* type for better performance.
- *Currency* is a fixed-point data type that minimizes rounding errors in monetary calculations. It is stored as a scaled 64-bit integer with the four least significant digits implicitly representing decimal places. When mixed with other real types in assignments and expressions, *Currency* values are automatically divided or multiplied by 10000.

String types

A string represents a sequence of characters. Object Pascal supports the following predefined string types.

Table 5.5 String types

Type	Maximum length	Memory required	Used for
ShortString	255 characters	2 to 256 bytes	backward compatibility
AnsiString	~2^{31} characters	4 bytes to 2GB	8-bit (ANSI) characters
WideString	~2^{30} characters	4 bytes to 2GB	Unicode characters; multi-user servers and multi-language applications

AnsiString, sometimes called the *long string*, is the preferred type for most purposes.

String types can be mixed in assignments and expressions; the compiler automatically performs required conversions. But strings passed by reference to a function or procedure (as **var** and **out** parameters) must be of the appropriate type. Strings can be explicitly cast to a different string type (see "Typecasts" on page 4-14).

The reserved word **string** functions like a generic type identifier. For example,

```
var S: string;
```

creates a variable *S* that holds a string. In the default **{$H+}** state, the compiler interprets **string** (when it appears without a bracketed number after it) as *AnsiString*. Use the **{$H–}** directive to turn **string** into *ShortString*.

The standard function *Length* returns the number of characters in a string. The *SetLength* procedure adjusts the length of a string. See the online Help for details.

Comparison of strings is defined by the ordering of the characters in corresponding positions. Between strings of unequal length, each character in the longer string without a corresponding character in the shorter string takes on a greater-than value. For example, "AB" is greater than "A"; that is, `'AB' > 'A'` returns *True*. Zero-length strings hold the lowest values.

You can index a string variable just as you would an array. If *S* is a string variable and *i* an integer expression, `S[i]` represents the *i*th character—or, strictly speaking, the *i*th byte—in *S*. For a *ShortString* or *AnsiString*, `S[i]` is of type *AnsiChar*; for a *WideString*, `S[i]` is of type *WideChar*. The statement `MyString[2] := 'A';` assigns the value *A* to the second character of *MyString*. The following code uses the standard *UpCase* function to convert *MyString* to uppercase.

```
var I: Integer;
begin
  I := Length(MyString);
  while I > 0 do
  begin
    MyString[I] := UpCase(MyString[I]);
    I := I - 1;
  end;
end;
```

Be careful indexing strings in this way, since overwriting the end of a string can cause access violations. Also, avoid passing long-string indexes as **var** parameters, because this results in inefficient code.

You can assign the value of a string constant—or any other expression that returns a string—to a variable. The length of the string changes dynamically when the assignment is made. Examples:

```
MyString := 'Hello world!';
MyString := 'Hello ' + 'world';
MyString := MyString + '!';
MyString := ' ';              { space }
MyString := '';               { empty string }
```

For more information, see "Character strings" on page 4-4 and "String operators" on page 4-9.

Short strings

A *ShortString* is 0 to 255 characters long. While the length of a *ShortString* can change dynamically, its memory is a statically allocated 256 bytes; the first byte stores the length of the string, and the remaining 255 bytes are available for characters. If *S* is a *ShortString* variable, `Ord(S[0])`, like `Length(S)`, returns the length of *S*; assigning a value to `S[0]`, like calling *SetLength*, changes the length of *S*. *ShortString* uses 8-bit ANSI characters and is maintained for backward compatibility only.

Object Pascal supports short-string types—in effect, subtypes of *ShortString*—whose maximum length is anywhere from 0 to 255 characters. These are denoted by a bracketed numeral appended to the reserved word **string**. For example,

```
var MyString: string[100];
```

creates a variable called *MyString* whose maximum length is 100 characters. This is equivalent to the declarations

```
type CString = string[100];
var MyString: CString;
```

Variables declared in this way allocate only as much memory as the type requires—that is, the specified maximum length plus one byte. In our example, *MyString* uses 101 bytes, as compared to 256 bytes for a variable of the predefined *ShortString* type.

When you assign a value to a short-string variable, the string is truncated if it exceeds the maximum length for the type.

The standard functions *High* and *Low* operate on short-string type identifiers and variables. *High* returns the maximum length of the short-string type, while *Low* returns zero.

Long strings

AnsiString, also called a *long string*, represents a dynamically allocated string whose maximum length is limited only by available memory. It uses 8-bit ANSI characters.

A long-string variable is a pointer occupying four bytes of memory. When the variable is empty—that is, when it contains a zero-length string—the pointer is **nil** and the string uses no additional storage. When the variable is nonempty, it points to a dynamically allocated block of memory that contains the string value, a 32-bit length indicator, and a 32-bit reference count. This memory is allocated on the heap, but its management is entirely automatic and requires no user code.

Because long-string variables are pointers, two or more of them can reference the same value without consuming additional memory. The compiler exploits this to conserve resources and execute assignments faster. Whenever a long-string variable is destroyed or assigned a new value, the reference count of the old string (the variable's previous value) is decremented and the reference count of the new value (if there is one) is incremented; if the reference count of a string reaches zero, its memory is deallocated. This process is called *reference-counting*. When indexing is used to change the value of a single character in a string, a copy of the string is made if—but only if—its reference count is greater than one. This is called *copy-on-write* semantics.

WideString

The *WideString* type represents a dynamically allocated string of 16-bit Unicode characters. In most respects it is similar to *AnsiString*.

On Win32, *WideString* is compatible with the COM *BSTR* type. Borland development tools have support features that convert *AnsiString* values to *WideString*, but you may need to explicitly cast or convert your strings to *WideString*.

About extended character sets

Windows and Linux both support *single-byte* and *multibyte* character sets as well as *Unicode*. With a single-byte character set (SBCS), each byte in a string represents one character. The ANSI character set used by many western operating systems is a single-byte character set.

In a multibyte character set (MBCS), some characters are represented by one byte and others by more than one byte. The first byte of a multibyte character is called the *lead byte*. In general, the lower 128 characters of a multibyte character set map to the 7-bit ASCII characters, and any byte whose ordinal value is greater than 127 is the lead byte of a multibyte character. Only single-byte characters can contain the null value (#0). Multibyte character sets—especially double-byte character sets (DBCS)—are widely used for Asian languages, while the UTF-8 character set used by Linux is a multibyte encoding of Unicode.

In the Unicode character set, each character is represented by two bytes. Thus a Unicode string is a sequence not of individual bytes but of two-byte words. Unicode characters and strings are also called *wide characters* and *wide character strings*. The first 256 Unicode characters map to the ANSI character set. The Windows operating system supports Unicode (UCS-2). The Linux operating system supports UCS-4, a superset of UCS-2. Delphi/Kylix supports UCS-2 on both platforms.

Object Pascal supports single-byte and multibyte characters and strings through the *Char*, *PChar*, *AnsiChar*, *PAnsiChar*, and *AnsiString* types. Indexing of multibyte strings is not reliable, since `S[i]` represents the *i*th byte (not necessarily the *i*th character) in *S*. However, the standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. (Names of multibyte functions usually start with *Ansi-*. For example, the multibyte version of *StrPos* is *AnsiStrPos*.) Multibyte character support is operating-system dependent and based on the current locale.

Object Pascal supports Unicode characters and strings through the *WideChar*, *PWideChar*, and *WideString* types.

Working with null-terminated strings

Many programming languages, including C and C++, lack a dedicated string data type. These languages, and environments that are built with them, rely on *null-terminated* strings. A null-terminated string is a zero-based array of characters that ends with NULL (#0); since the array has no length indicator, the first NULL character marks the end of the string. You can use Object Pascal constructions and special routines in the *SysUtils* unit (see Chapter 8, “Standard routines and I/O”) to handle null-terminated strings when you need to share data with systems that use them.

For example, the following type declarations could be used to store null-terminated strings.

```
type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..259] of Char;
  TMemoText = array[0..1023] of WideChar;
```

With extended syntax enabled (**{$X+}**), you can assign a string constant to a statically allocated zero-based character array. (Dynamic arrays won’t work for this purpose.) If you initialize an array constant with a string that is shorter than the declared length of the array, the remaining characters are set to #0. For more information about arrays, see “Arrays” on page 5-18.

Using pointers, arrays, and string constants

To manipulate null-terminated strings, it is often necessary to use pointers. (See “Pointers and pointer types” on page 5-25.) String constants are assignment-compatible with the *PChar* and *PWideChar* types, which represent pointers to null-terminated arrays of *Char* and *WideChar* values. For example,

```
var P: PChar;
  ⋮
P := 'Hello world!';
```

points *P* to an area of memory that contains a null-terminated copy of “Hello world!” This is equivalent to

```
const TempString: array[0..12] of Char = 'Hello world!'#0;
var P: PChar;
  ⋮
P := @TempString;
```

You can also pass string constants to any function that takes value or **const** parameters of type *PChar* or *PWideChar*—for example `StrUpper('Hello world!')`. As with assignments to a *PChar*, the compiler generates a null-terminated copy of the string and gives the function a pointer to that copy. Finally, you can initialize *PChar* or *PWideChar* constants with string literals, alone or in a structured type. Examples:

```
const
Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits: array[0..9] of PChar = (
    'Zero', 'One', 'Two', 'Three', 'Four',
    'Five', 'Six', 'Seven', 'Eight', 'Nine');
```

Zero-based character arrays are compatible with *PChar* and *PWideChar*. When you use a character array in place of a pointer value, the compiler converts the array to a pointer constant whose value corresponds to the address of the first element of the array. For example,

```
var
  MyArray: array[0..32] of Char;
  MyPointer: PChar;
begin
  MyArray := 'Hello';
  MyPointer := MyArray;
  SomeProcedure(MyArray);
  SomeProcedure(MyPointer);
end;
```

This code calls *SomeProcedure* twice with the same value.

A character pointer can be indexed as if it were an array. In the example above, `MyPointer[0]` returns *H*. The index specifies an offset added to the pointer before it is dereferenced. (For *PWideChar* variables, the index is automatically multiplied by two.) Thus, if *P* is a character pointer, *P*`[0]` is equivalent to *P*^ and specifies the first character in the array, *P*`[1]` specifies the second character in the array, and so forth; *P*`[-1]` specifies the "character" immediately to the left of *P*`[0]`. *The compiler performs no range checking on these indexes.*

The *StrUpper* function illustrates the use of pointer indexing to iterate through a null-terminated string:

```
function StrUpper(Dest, Source: PChar; MaxLen: Integer): PChar;
var
  I: Integer;
begin
  I := 0;
  while (I < MaxLen) and (Source[I] <> #0) do
  begin
    Dest[I] := UpCase(Source[I]);
    Inc(I);
  end;
  Dest[I] := #0;
  Result := Dest;
end;
```

Mixing Pascal strings and null-terminated strings

You can mix long strings (*AnsiString* values) and null-terminated strings (*PChar* values) in expressions and assignments, and you can pass *PChar* values to functions or procedures that take long-string parameters. The assignment *S* `:=` *P*, where *S* is a string variable and *P* is a *PChar* expression, copies a null-terminated string into a long string.

In a binary operation, if one operand is a long string and the other a *PChar*, the *PChar* operand is converted to a long string.

You can cast a *PChar* value as a long string. This is useful when you want to perform a string operation on two *PChar* values. For example,

```
S := string(P1) + string(P2);
```

You can also cast a long string as a null-terminated string. The following rules apply.

- If *S* is a long-string expression, `PChar(S)` casts *S* as a null-terminated string; it returns a pointer to the first character in *S*.

  On Windows: For example, if *Str1* and *Str2* are long strings, you could call the Win32 API *MessageBox* function like this:

  ```
  MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);
  ```

  (The declaration of *MessageBox* is in the *Windows* interface unit.)

  On Linux: For example, if *Str* is a long string, you could call the *opendir* system function like this:

  ```
  opendir(PChar(Str));
  ```

  (The declaration of *opendir* is in the *Libc* interface unit.)

- You can also use `Pointer(S)` to cast a long string to an untyped pointer. But if *S* is empty, the typecast returns **nil**.
- When you cast a long-string variable to a pointer, the pointer remains valid until the variable is assigned a new value or goes out of scope. If you cast any other long-string expression to a pointer, the pointer is valid only within the statement where the typecast is performed.
- When you cast a long-string expression to a pointer, the pointer should usually be considered read-only. You can safely use the pointer to modify the long string only when all of the following conditions are satisfied.
  - The expression cast is a long-string *variable*.
  - The string is not empty.
  - The string is unique—that is, has a reference count of one. To guarantee that the string is unique, call the *SetLength*, *SetString*, or *UniqueString* procedure.
  - The string has not been modified since the typecast was made.
  - The characters modified are all within the string. Be careful not to use an out-of-range index on the pointer.

The same rules apply when mixing *WideString* values with *PWideChar* values.

Structured types

Instances of a structured type hold more than one value. Structured types include *sets*, *arrays*, *records*, and *files* as well as *class*, *class-reference*, and *interface* types. (For information about class and class-reference types, see Chapter 7, "Classes and objects." For information about interfaces, see Chapter 10, "Object interfaces".) Except for sets, which hold ordinal values only, structured types can contain other structured types; a type can have unlimited levels of structuring.

By default, the values in a structured type are aligned on word or double-word boundaries for faster access. When you declare a structured type, you can include the reserved word **packed** to implement compressed data storage. For example,

```
type TNumbers = packed array[1..100] of Real;
```

Using **packed** slows data access and, in the case of a character array, affects type compatibility. For more information, see Chapter 11, "Memory management".

Sets

A set is a collection of values of the same ordinal type. The values have no inherent order, nor is it meaningful for a value to be included twice in a set.

The range of a set type is the power set of a specific ordinal type, called the *base type*; that is, the possible values of the set type are all the subsets of the base type, including the empty set. The base type can have no more than 256 possible values, and their ordinalities must fall between 0 and 255. Any construction of the form

```
set of baseType
```

where *baseType* is an appropriate ordinal type, identifies a set type.

Because of the size limitations for base types, set types are usually defined with subranges. For example, the declarations

```
type
  TSomeInts = 1..250;
  TIntSet = set of TSomeInts;
```

create a set type called *TIntSet* whose values are collections of integers in the range from 1 to 250. You could accomplish the same thing with

```
type TIntSet = set of 1..250;
```

Given this declaration, you can create a sets like this:

```
var Set1, Set2: TIntSet;
 ⋮
Set1 := [1, 3, 5, 7, 9];
Set2 := [2, 4, 6, 8, 10]
```

You can also use the **set of ...** construction directly in variable declarations:

```
var MySet: set of 'a'..'z';
 ⋮
MySet := ['a','b','c'];
```

Other examples of set types include

```
set of Byte
set of (Club, Diamond, Heart, Spade)
set of Char;
```

The **in** operator tests set membership:

```
if 'a' in MySet then ... { do something } ;
```

Every set type can hold the empty set, denoted by `[]`. For more information about sets, see "Set constructors" on page 4-13 and "Set operators" on page 4-10.

Arrays

An array represents an indexed collection of elements of the same type (called the *base type*). Because each element has a unique index, arrays, unlike sets, can meaningfully contain the same value more than once. Arrays can be allocated *statically* or *dynamically*.

Static arrays

Static array types are denoted by constructions of the form

`array[`*indexType*$_1$`, ..., `*indexType*$_n$`] of` *baseType*

where each *indexType* is an ordinal type whose range does not exceed 2GB. Since the *indexType*s index the array, the number of elements an array can hold is limited by the product of the sizes of the *indexType*s. In practice, *indexType*s are usually integer subranges.

In the simplest case of a one-dimensional array, there is only a single *indexType*. For example,

```
var MyArray: array[1..100] of Char;
```

declares a variable called *MyArray* that holds an array of 100 character values. Given this declaration, `MyArray[3]` denotes the third character in *MyArray*. If you create a static array but don't assign values to all its elements, the unused elements are still allocated and contain random data; they are like uninitialized variables.

A multidimensional array is an array of arrays. For example,

```
type TMatrix = array[1..10] of array[1..50] of Real;
```

is equivalent to

```
type TMatrix = array[1..10, 1..50] of Real;
```

Whichever way *TMatrix* is declared, it represents an array of 500 real values. A variable *MyMatrix* of type *TMatrix* can be indexed like this: `MyMatrix[2,45]`; or like this: `MyMatrix[2][45]`. Similarly,

```
packed array[Boolean,1..10,TShoeSize] of Integer;
```

is equivalent to

```
packed array[Boolean] of packed array[1..10] of packed array[TShoeSize] of Integer;
```

The standard functions *Low* and *High* operate on array type identifiers and variables. They return the low and high bounds of the array's first index type. The standard function *Length* returns the number of elements in the array's first dimension.

A one-dimensional, packed, static array of *Char* values is called a *packed string*. Packed-string types are compatible with string types and with other packed-string types that have the same number of elements. See "Type compatibility and identity" on page 5-35.

An array type of the form `array[0..`*x*`] of Char` is called a *zero-based character array*. Zero-based character arrays are used to store null-terminated strings and are compatible with *PChar* values. See "Working with null-terminated strings" on page 5-13.

Dynamic arrays

Dynamic arrays do not have a fixed size or length. Instead, memory for a dynamic array is reallocated when you assign a value to the array or pass it to the *SetLength* procedure. Dynamic-array types are denoted by constructions of the form

```
array of baseType
```

For example,

```
var MyFlexibleArray: array of Real;
```

declares a one-dimensional dynamic array of reals. The declaration does not allocate memory for *MyFlexibleArray*. To create the array in memory, call *SetLength*. For example, given the declaration above,

```
SetLength(MyFlexibleArray, 20);
```

allocates an array of 20 reals, indexed 0 to 19. Dynamic arrays are always integer-indexed, always starting from 0.

Dynamic-array variables are implicitly pointers and are managed by the same reference-counting technique used for long strings. To deallocate a dynamic array, assign **nil** to a variable that references the array or pass the variable to *Finalize*; either of these methods disposes of the array, provided there are no other references to it. Dynamic arrays of length 0 have the value **nil**. Do not apply the dereference operator (^) to a dynamic-array variable or pass it to the *New* or *Dispose* procedure.

If *X* and *Y* are variables of the same dynamic-array type, *X* `:=` *Y* points *X* to the same array as *Y*. (There is no need to allocate memory for *X* before performing this operation.) Unlike strings and static arrays, dynamic arrays are not automatically copied before they are written to. For example, after this code executes—

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := A;
  B[0] := 2;
end;
```

—the value of `A[0]` is 2. (If *A* and *B* were static arrays, `A[0]` would still be 1.)

Assigning to a dynamic-array index (for example, `MyFlexibleArray[2] := 7`) does not reallocate the array. Out-of-range indexes are not reported at compile time.

In contrast, to make an independent copy of a dynamic array, you must use the global Copy function:

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := Copy(A);
  B[0] := 2; { B[0] <> A[0] }
end;
```

When dynamic-array variables are compared, their references are compared, not their array values. Thus, after execution of the code

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  SetLength(B, 1);
  A[0] := 2;
  B[0] := 2;
end;
```

`A = B` returns *False* but `A[0] = B[0]` returns *True*.

To truncate a dynamic array, pass it to *SetLength* or *Copy* and assign the result back to the array variable. (The *SetLength* procedure is usually faster.) For example, if *A* is a dynamic array, `A := SetLength(A, 0, 20)` truncates all but the first 20 elements of *A*.

Once a dynamic array has been allocated, you can pass it to the standard functions *Length*, *High*, and *Low*. *Length* returns the number of elements in the array, *High* returns the array's highest index (that is, *Length*–1), and *Low* returns 0. In the case of a zero-length array, *High* returns –1 (with the anomalous consequence that *High* < *Low*).

Note In some function and procedure declarations, array parameters are represented as `array of` *baseType*, without any index types specified. For example,

```
function CheckStrings(A: array of string): Boolean;
```

This indicates that the function operates on all arrays of the specified base type, regardless of their size, how they are indexed, or whether they are allocated statically or dynamically. See "Open array parameters" on page 6-15.

Multidimensional dynamic arrays

To declare multidimensional dynamic arrays, use iterated **array of ...** constructions. For example,

```
type TMessageGrid = array of array of string;
var Msgs: TMessageGrid;
```

declares a two-dimensional array of strings. To instantiate this array, call *SetLength* with two integer arguments. For example, if *I* and *J* are integer-valued variables,

```
SetLength(Msgs,I,J);
```

allocates an *I*-by-*J* array, and `Msgs[0,0]` denotes an element of that array.

You can create multidimensional dynamic arrays that are not rectangular. The first step is to call *SetLength*, passing it parameters for the first *n* dimensions of the array. For example,

```
var Ints: array of array of Integer;
SetLength(Ints,10);
```

allocates ten rows for *Ints* but no columns. Later, you can allocate the columns one at a time (giving them different lengths); for example

```
SetLength(Ints[2], 5);
```

makes the third column of *Ints* five integers long. At this point (even if the other columns haven't been allocated) you can assign values to the third column—for example, `Ints[2,4] := 6`.

The following example uses dynamic arrays (and the *IntToStr* function declared in the *SysUtils* unit) to create a triangular matrix of strings.

```
var
  A : array of array of string;
  I, J : Integer;
begin
  SetLength(A, 10);
  for I := Low(A) to High(A) do
  begin
    SetLength(A[I], I);
    for J := Low(A[I]) to High(A[I]) do
      A[I,J] := IntToStr(I) + ',' + IntToStr(J) + ' ';
  end;
end;
```

Array types and assignments

Arrays are assignment-compatible only if they are of the same type. Because Pascal uses name-equivalence for types, the following code will not compile.

```
var
  Int1: array[1..10] of Integer;
  Int2: array[1..10] of Integer;
  ⋮
Int1 := Int2;
```

To make the assignment work, declare the variables as

```
var Int1, Int2: array[1..10] of Integer;
```

or

```
type IntArray = array[1..10] of Integer;
var
  Int1: IntArray;
  Int2: IntArray;
```

Records

A record (analogous to a *structure* in some languages) represents a heterogeneous set of elements. Each element is called a *field*; the declaration of a record type specifies a name and type for each field. The syntax of a record type declaration is

```
type recordTypeName = record
  fieldList1: type1;
  ⋮
  fieldListn: typen;
end
```

where *recordTypeName* is a valid identifier, each *type* denotes a type, and each *fieldList* is a valid identifier or a comma-delimited list of identifiers. The final semicolon is optional.

For example, the following declaration creates a record type called *TDateRec*.

```
type
  TDateRec = record
    Year: Integer;
    Month: (Jan, Feb, Mar, Apr, May, Jun,
            Jul, Aug, Sep, Oct, Nov, Dec);
    Day: 1..31;
  end;
```

Each *TDateRec* contains three fields: an integer value called *Year*, a value of an enumerated type called *Month*, and another integer between 1 and 31 called *Day*. The identifiers *Year*, *Month*, and *Day* are the *field designators* for *TDateRec*, and they behave like variables. The *TDateRec* type declaration, however, does not allocate any memory for the *Year*, *Month*, and *Day* fields; memory is allocated when you instantiate the record, like this:

```
var Record1, Record2: TDateRec;
```

This variable declaration creates two instances of *TDateRec*, called *Record1* and *Record2*.

You can access the fields of a record by qualifying the field designators with the record's name:

```
Record1.Year := 1904;
Record1.Month := Jun;
Record1.Day := 16;
```

Or use a **with** statement:

```
with Record1 do
begin
  Year := 1904;
  Month := Jun;
  Day := 16;
end;
```

You can now copy the values of *Record1*'s fields to *Record2*:

```
Record2 := Record1;
```

Because the scope of a field designator is limited to the record in which it occurs, you don't have to worry about naming conflicts between field designators and other variables.

Instead of defining record types, you can use the **record ...** construction directly in variable declarations:

```
var S: record
  Name: string;
  Age: Integer;
end;
```

However, a declaration like this largely defeats the purpose of records, which is to avoid repetitive coding of similar groups of variables. Moreover, separately declared records of this kind will not be assignment-compatible, even if their structures are identical.

Variant parts in records

A record type can have a *variant* part, which looks like a **case** statement. The variant part must follow the other fields in the record declaration.

To declare a record type with a variant part, use the following syntax.

```
type recordTypeName = record
  fieldList1: type1;
  ⋮
  fieldListn: typen;
case tag: ordinalType of
  constantList1: (variant1);
  ⋮
  constantListn: (variantn);
end;
```

The first part of the declaration—up to the reserved word **case**—is the same as that of a standard record type. The remainder of the declaration—from **case** to the optional final semicolon—is called the variant part. In the variant part,

- *tag* is optional and can be any valid identifier. If you omit *tag*, omit the colon (**:**) after it as well.
- *ordinalType* denotes an ordinal type.
- Each *constantList* is a constant denoting a value of type o*rdinalType*, or a comma-delimited list of such constants. No value can be represented more than once in the combined *constantList*s.
- Each *variant* is a comma-delimited list of declarations resembling the *fieldList*: *type* constructions in the main part of the record type. That is, a *variant* has the form

  ```
  fieldList1: type1;
    ⋮
  fieldListn: typen;
  ```

  where each *fieldList* is a valid identifier or comma-delimited list of identifiers, each *type* denotes a type, and the final semicolon is optional. The *types* must not be long strings, dynamic arrays, variants (that is, *Variant* types), or interfaces, nor can they be structured types that contain long strings, dynamic arrays, variants, or interfaces; but they can be pointers to these types.

Records with variant parts are complicated syntactically but deceptively simple semantically. The variant part of a record contains several *variant*s which share the same space in memory. You can read or write to any field of any *variant* at any time; but if you write to a field in one *variant* and then to a field in another *variant*, you may be overwriting your own data. The *tag*, if there is one, functions as an extra field (of type *ordinalType*) in the non-variant part of the record.

Variant parts have two purposes. First, suppose you want to create a record type that has fields for different kinds of data, but you know that you will never need to use all of the fields in a single record instance. For example,

```
type
  TEmployee = record
  FirstName, LastName: string[40];
  BirthDate: TDate;
  case Salaried: Boolean of
    True: (AnnualSalary: Currency);
    False: (HourlyWage: Currency);
end;
```

The idea here is that every employee has either a salary or an hourly wage, but not both. So when you create an instance of *TEmployee*, there is no reason to allocate enough memory for both fields. In this case, the only difference between the *variant*s is in the field names, but the fields could just as easily have been of different types. Consider some more complicated examples:

```
type
  TPerson = record
  FirstName, LastName: string[40];
  BirthDate: TDate;
  case Citizen: Boolean of
    True: (Birthplace: string[40]);
    False: (Country: string[20];
            EntryPort: string[20];
            EntryDate, ExitDate: TDate);
  end;

type
  TShapeList = (Rectangle, Triangle, Circle, Ellipse, Other);
  TFigure = record
    case TShapeList of
      Rectangle: (Height, Width: Real);
      Triangle: (Side1, Side2, Angle: Real);
      Circle: (Radius: Real);
      Ellipse, Other: ();
  end;
```

For each record instance, the compiler allocates enough memory to hold all the fields in the largest *variant*. The optional *tag* and the *constantList*s (like *Rectangle*, *Triangle*, and so forth in the last example above) play no role in the way the compiler manages the fields; they are there only for the convenience of the programmer.

The second reason for variant parts is that they let you treat the same data as belonging to different types, even in cases where the compiler would not allow a typecast. For example, if you have a 64-bit *Real* as the first field in one *variant* and a 32-bit *Integer* as the first field in another, you can assign a value to the *Real* field and then read back the first 32 bits of it as the value of the *Integer* field (passing it, say, to a function that requires integer parameters).

File types

A file is an ordered set of elements of the same type. Standard I/O routines use the predefined *TextFile* or *Text* type, which represents a file containing characters organized into lines. For more information about file input and output, see Chapter 8, “Standard routines and I/O”.

To declare a file type, use the syntax

`type` *fileTypeName* `= file of` *type*

where *fileTypeName* is any valid identifier and *type* is a fixed-size type. Pointer types—whether implicit or explicit—are not allowed, so a file cannot contain dynamic arrays, long strings, classes, objects, pointers, variants, other files, or structured types that contain any of these.

For example,

```
type
  PhoneEntry = record
    FirstName, LastName: string[20];
    PhoneNumber: string[15];
    Listed: Boolean;
  end;
  PhoneList = file of PhoneEntry;
```

declares a file type for recording names and telephone numbers.

You can also use the **file of ...** construction directly in a variable declaration. For example,

```
var List1: file of PhoneEntry;
```

The word **file** by itself indicates an untyped file:

```
var DataFile: file;
```

For more information, see “Untyped files” on page 8-4.

Files are not allowed in arrays or records.

Pointers and pointer types

A pointer is a variable that denotes a memory address. When a pointer holds the address of another variable, we say that it *points* to the location of that variable in memory or to the data stored there. In the case of an array or other structured type, a pointer holds the address of the first element in the structure.

Pointers are *typed* to indicate the kind of data stored at the addresses they hold. The general-purpose *Pointer* type can represent a pointer to any data, while more specialized pointer types reference only specific types of data. Pointers occupy four bytes of memory.

Overview of pointers

To see how pointers work, look at the following example.

```
1    var
2      X, Y: Integer;   // X and Y are Integer variables
3      P: ^Integer;     // P points to an Integer
4    begin
5      X := 17;         // assign a value to X
6      P := @X;         // assign the address of X to P
7      Y := P^;         // dereference P; assign the result to Y
8    end;
```

Line 2 declares *X* and *Y* as variables of type *Integer*. Line 3 declares *P* as a pointer to an *Integer* value; this means that *P* can point to the location of *X* or *Y*. Line 5 assigns a value to *X*, and line 6 assigns the address of *X* (denoted by `@X`) to *P*. Finally, line 7 retrieves the value at the location pointed to by *P* (denoted by `^P`) and assigns it to *Y*. After this code executes, *X* and *Y* have the same value, namely 17.

The @ operator, which we have used here to take the address of a variable, also operates on functions and procedures. For more information, see "The @ operator" on page 4-12 and "Procedural types in statements and expressions" on page 5-30.

The symbol **^** has two purposes, both of which are illustrated in our example. When it appears before a type identifier—

^*typeName*

—it denotes a type that represents pointers to variables of type *typeName*. When it appears after a pointer variable—

pointer^

—it *dereferences* the pointer; that is, it returns the value stored at the memory address held by the pointer.

Our example may seem like a roundabout way of copying the value of one variable to another—something that we could have accomplished with a simple assignment statement. But pointers are useful for several reasons. First, understanding pointers will help you to understand Object Pascal, since pointers often operate behind the scenes in code where they don't appear explicitly. Any data type that requires large, dynamically allocated blocks of memory uses pointers. Long-string variables, for instance, are implicitly pointers, as are class variables. Moreover, some advanced programming techniques require the use of pointers.

Finally, pointers are sometimes the only way to circumvent Object Pascal's strict data typing. By referencing a variable with an all-purpose *Pointer*, casting the *Pointer* to a more specific type, and then dereferencing it, you can treat the data stored by any variable as if it belonged to any type. For example, the following code assigns data stored in a real variable to an integer variable.

```
type
  PInteger = ^Integer;
var
  R: Single;
```

```
    I: Integer;
    P: Pointer;
    PI: PInteger;
  begin
    ⋮
    P := @R;
    PI := PInteger(P);
    I := PI^;
  end;
```

Of course, reals and integers are stored in different formats. This assignment simply copies raw binary data from *R* to *I*, without converting it.

In addition to assigning the result of an @ operation, you can use several standard routines to give a value to a pointer. The *New* and *GetMem* procedures assign a memory address to an existing pointer, while the *Addr* and *Ptr* functions return a pointer to a specified address or variable.

Dereferenced pointers can be qualified and can function as qualifiers, as in the expression `P1^.Data^`.

The reserved word **nil** is a special constant that can be assigned to any pointer. When **nil** is assigned to a pointer, the pointer doesn't reference anything.

Pointer types

You can declare a pointer to any type, using the syntax

`type` *pointerTypeName* `=` `^`*type*

When you define a record or other data type, it's a common practice also to define a pointer to that type. This makes it easy to manipulate instances of the type without copying large blocks of memory.

Standard pointer types exist for many purposes. The most versatile is *Pointer*, which can point to data of any kind. But a *Pointer* variable cannot be dereferenced; placing the **^** symbol after a *Pointer* variable causes a compilation error. To access the data referenced by a *Pointer* variable, first cast it to another pointer type and then dereference it.

Character pointers

The fundamental types *PAnsiChar* and *PWideChar* represent pointers to *AnsiChar* and *WideChar* values, respectively. The generic *PChar* represents a pointer to a *Char* (that is, in its current implementation, to an *AnsiChar*). These character pointers are used to manipulate null-terminated strings. (See "Working with null-terminated strings" on page 5-13.)

Other standard pointer types

The *System* and *SysUtils* units declare many standard pointer types that are commonly used.

Table 5.6 Selected pointer types declared in System and SysUtils

Pointer type	Points to variables of type
PAnsiString, *PString*	*AnsiString*
PByteArray	*TByteArray* (declared in *SysUtils*). Used to typecast dynamically allocated memory for array access.
PCurrency, *PDouble*, *PExtended*, *PSingle*	*Currency*, *Double*, *Extended*, *Single*
PInteger	*Integer*
POleVariant	*OleVariant*
PShortString	*ShortString*. Useful when porting legacy code that uses the old *PString* type.
PTextBuf	*TTextBuf* (declared in *SysUtils*). *TTextBuf* is the internal buffer type in a *TTextRec* file record.)
PVarRec	*TVarRec* (declared in *System*)
PVariant	*Variant*
PWideString	*WideString*
PWordArray	*TWordArray* (declared in *SysUtils*). Used to typecast dynamically allocated memory for arrays of 2-byte values.

Procedural types

Procedural types allow you to treat procedures and functions as values that can be assigned to variables or passed to other procedures and functions. For example, suppose you define a function called *Calc* that takes two integer parameters and returns an integer:

```
function Calc(X,Y: Integer): Integer;
```

You can assign the *Calc* function to the variable *F*:

```
var F: function(X,Y: Integer): Integer;
F := Calc;
```

If you take any procedure or function heading and remove the identifier after the word **procedure** or **function**, what's left is the name of a procedural type. You can use such type names directly in variable declarations (as in the example above) or to declare new types:

```
type
  TIntegerFunction = function: Integer;
  TProcedure = procedure;
  TStrProc = procedure(const S: string);
  TMathFunc = function(X: Double): Double;
var
  F: TIntegerFunction;              { F is a parameterless function that returns an integer }
  Proc: TProcedure;              { Proc is a parameterless procedure }
  SP: TStrProc;                  { SP is a procedure that takes a string parameter }
  M: TMathFunc;                  { M is a function that takes a Double (real) parameter
                                     and returns a Double }
procedure FuncProc(P: TIntegerFunction);  { FuncProc is a procedure whose only parameter
                                             is a parameterless integer-valued function }
```

The variables above are all *procedure pointers*—that is, pointers to the address of a procedure or function. If you want to reference a method of an instance object (see Chapter 7, "Classes and objects"), you need to add the words **of object** to the procedural type name. For example

```
type
  TMethod = procedure of object;
  TNotifyEvent = procedure(Sender: TObject) of object;
```

These types represent *method pointers*. A method pointer is really a pair of pointers; the first stores the address of a method, and the second stores a reference to the object the method belongs to. Given the declarations

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
  TMainForm = class(TForm)
    procedure ButtonClick(Sender: TObject);
    ⋮
  end;
var
  MainForm: TMainForm;
  OnClick: TNotifyEvent
```

we could make the following assignment.

```
OnClick := MainForm.ButtonClick;
```

Two procedural types are compatible if they have

- the same calling convention,
- the same return value (or no return value), and
- the same number of parameters, with identically typed parameters in corresponding positions. (Parameter names do not matter.)

Procedure pointer types are always incompatible with method pointer types. The value **nil** can be assigned to any procedural type.

Nested procedures and functions (routines declared within other routines) cannot be used as procedural values, nor can predefined procedures and functions. If you want to use a predefined routine like *Length* as a procedural value, write a wrapper for it:

```
function FLength(S: string): Integer;
begin
  Result := Length(S);
end;
```

Procedural types in statements and expressions

When a procedural variable is on the left side of an assignment statement, the compiler expects a procedural value on the right. The assignment makes the variable on the left a pointer to the function or procedure indicated on the right. In other contexts, however, using a procedural variable results in a call to the referenced procedure or function. You can even use a procedural variable to pass parameters:

```
var
  F: function(X: Integer): Integer;
  I: Integer;
function SomeFunction(X: Integer): Integer;
 ⋮
F := SomeFunction;  // assign SomeFunction to F
I := F(4);          // call function; assign result to I
```

In assignment statements, the type of the variable on the left determines the interpretation of procedure or method pointers on the right. For example,

```
var
  F, G: function: Integer;
  I: Integer;
function SomeFunction: Integer;
 ⋮
F := SomeFunction;  // assign SomeFunction to F
G := F;             // copy F to G
I := G;             // call function; assign result to I
```

The first statement assigns a procedural value to *F*. The second statement copies that value to another variable. The third statement makes a call to the referenced function and assigns the result to *I*. Because *I* is an integer variable, not a procedural one, the last assignment actually calls the function (which returns an integer).

In some situations it is less clear how a procedural variable should be interpreted. Consider the statement

```
if F = MyFunction then ...;
```

In this case, the occurrence of *F* results in a function call; the compiler calls the function pointed to by *F*, then calls the function *MyFunction*, then compares the results. The rule is that whenever a procedural variable occurs within an expression, it represents a call to the referenced procedure or function. In a case where *F* references a procedure (which doesn't return a value), or where *F* references a function that requires parameters, the statement above causes a compilation error. To compare the procedural value of *F* with *MyFunction*, use

```
if @F = @MyFunction then ...;
```

`@F` converts *F* into an untyped pointer variable that contains an address, and `@MyFunction` returns the address of *MyFunction*.

To get the memory address of a procedural variable (rather than the address stored in it), use @@. For example, `@@F` returns the address of *F*.

The @ operator can also be used to assign an untyped pointer value to a procedural variable. For example,

```
var StrComp: function(Str1, Str2: PChar): Integer;
 ⋮
@StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');
```

calls the *GetProcAddress* function and points *StrComp* to the result.

Any procedural variable can hold the value **nil**, which means that it points to nothing. But attempting to *call* a **nil**-valued procedural variable is an error. To test whether a procedural variable is assigned, use the standard function *Assigned*:

```
if Assigned(OnClick) then OnClick(X);
```

Variant types

Sometimes it is necessary to manipulate data whose type varies or cannot be determined at compile time. In these cases, one option is to use variables and parameters of type *Variant*, which represent values that can change type at runtime. Variants offer greater flexibility but consume more memory than regular variables, and operations on them are slower than on statically bound types. Moreover, illicit operations on variants often result in runtime errors, where similar mistakes with regular variables would have been caught at compile time. You can also create custom variant types.

By default, Variants can hold values of any type except records, sets, static arrays, files, classes, class references, and pointers. In other words, variants can hold anything but structured types and pointers. They can hold interfaces, whose methods and properties can be accessed through them. (See Chapter 10, "Object interfaces".) They can hold dynamic arrays, and they can hold a special kind of static array called a *variant array*. (See "Variant arrays" on page 5-34.) Variants can mix with other variants and with integer, real, string, and Boolean values in expressions and assignments; the compiler automatically performs type conversions.

Variants that contain strings cannot be indexed. That is, if *V* is a variant that holds a string value, the construction `V[1]` causes a runtime error.

You can define custom Variants that extend the Variant type to hold arbitrary values. For example, you can define a Variant string type that allows indexing or that holds a particular class reference, record type, or static array. Custom Variant types are defined by creating descendants to the *TCustomVariantType* class.

A variant occupies 16 bytes of memory and consists of a type code and a value, or pointer to a value, of the type specified by the code. All variants are initialized on creation to the special value *Unassigned*. The special value *Null* indicates unknown or missing data.

The standard function *VarType* returns a variant's type code. The *varTypeMask* constant is a bit mask used to extract the code from *VarType*'s return value, so that, for example,

```
VarType(V) and varTypeMask = varDouble
```

returns *True* if *V* contains a *Double* or an array of *Double*. (The mask simply hides the first bit, which indicates whether the variant holds an array.) The *TVarData* record type defined in the *System* unit can be used to typecast variants and gain access to their internal representation. See the online Help on *VarType* for a list if codes, and note that new type codes may be added in future implementations of Object Pascal.

Variant type conversions

All integer, real, string, character, and Boolean types are assignment-compatible with *Variant*. Expressions can be explicitly cast as variants, and the *VarAsType* and *VarCast* standard routines can be used to change the internal representation of a variant. The following code demonstrates the use of variants and some of the automatic conversions performed when variants are mixed with other types.

```
var
  V1, V2, V3, V4, V5: Variant;
  I: Integer;
  D: Double;
  S: string;
begin
  V1 := 1; { integer value }
  V2 := 1234.5678; { real value }
  V3 := 'Hello world!'; { string value }
  V4 := '1000'; { string value }
  V5 := V1 + V2 + V4; { real value 2235.5678}
  I := V1; { I = 1 (integer value) }
  D := V2; { D = 1234.5678 (real value) }
  S := V3; { S = 'Hello world!' (string value) }
  I := V4; { I = 1000 (integer value) }
  S := V5; { S = '2235.5678' (string value) }
end;
```

The compiler performs type conversions according to the following rules.

Table 5.7 Variant type conversion rules

Target Source	**integer**	**real**	**string**	**character**	**Boolean**
integer	converts integer formats	converts to real	converts to string representation	same as string (left)	returns *False* if 0, *True* otherwise
real	rounds to nearest integer	converts real formats	converts to string representation using regional settings	same as string (left)	returns *False* if 0, *True* otherwise

Table 5.7 Variant type conversion rules (continued)

string	converts to integer, truncating if necessary; raises exception if string is not numeric	converts to real using regional settings; raises exception if string is not numeric	converts string/ character formats	same as string (left)	returns *False* if string is “false” (non–case-sensitive) or a numeric string that evaluates to 0, *True* if string is “true” or a nonzero numeric string; raises exception otherwise
character	same as string (above)	same as string (above)	same as string (above)	same as string-to-string	same as string (above)
Boolean	*False* = 0, *True* = –1 (255 if *Byte*)	*False* = 0, *True* = –1	*False* = “0”, *True* = “–1”	same as string (left)	*False* = *False*, *True* = *True*
Unassigned	returns 0	returns 0	returns empty string	same as string (left)	returns *False*
Null	raises exception	raises exception	raises exception	same as string (left)	raises exception

Out-of-range assignments often result in the target variable getting the highest value in its range. Invalid assignments or casts raise the *EVariantError* exception.

Special conversion rules apply to the *TDateTime* real type declared in the *System* unit. When a *TDateTime* is converted to any other type, it treated as a normal *Double*. When an integer, real, or Boolean is converted to a *TDateTime*, it is first converted to a *Double*, then read as a date-time value. When a string is converted to a *TDateTime*, it is interpreted as a date-time value using the regional settings. When an *Unassigned* value is converted to *TDateTime*, it is treated like the real or integer value 0. Converting a *Null* value to *TDateTime* raises an exception.

On Windows, if a variant references a COM interface, any attempt to convert it reads the object’s default property and converts that value to the requested type. If the object has no default property, an exception is raised.

Variants in expressions

All operators except **^**, **is**, and **in** take variant operands. Operations on variants return *Variant* values; they return *Null* if one or both operands is *Null*, and raise an exception if one or both operands is *Unassigned*. In a binary operation, if only one operand is a variant, the other is converted to a variant.

The return type of an operation is determined by its operands. In general, the same rules that apply to operands of statically bound types apply to variants. For example, if *V1* and *V2* are variants that hold an integer and a real value, then `V1 + V2` returns a real-valued variant. (See "Operators" on page 4-6.) With variants, however, you can perform binary operations on combinations of values that would not be allowed using statically typed expressions. When possible, the compiler converts mismatched variants using the rules summarized in Table 5.7. For example, if *V3* and *V4* are variants that hold a numeric string and an integer, the expression `V3 + V4` returns an integer-valued variant; the numeric string is converted to an integer before the operation is performed.

Variant arrays

You cannot assign an ordinary static array to a variant. Instead, create a *variant array* by calling either of the standard functions *VarArrayCreate* or *VarArrayOf*. For example,

```
V: Variant;
 ⋮
V := VarArrayCreate([0,9], varInteger);
```

creates a variant array of integers (of length 10) and assigns it to the variant *V*. The array can be indexed using `V[0]`, `V[1]`, and so forth, but it is not possible to pass a variant array element as a **var** parameter. Variant arrays are always indexed with integers.

The second parameter in the call to *VarArrayCreate* is the type code for the array's base type. For a list of these codes, see the online Help on *VarType*. Never pass the code *varString* to *VarArrayCreate*; to create a variant array of strings, use *varOleStr*.

Variants can hold variant arrays of different sizes, dimensions, and base types. The elements of a variant array can be of any type allowed in variants except *ShortString* and *AnsiString*, and if the base type of the array is *Variant*, its elements can even be heterogeneous. Use the *VarArrayRedim* function to resize a variant array. Other standard routines that operate on variant arrays include *VarArrayDimCount*, *VarArrayLowBound*, *VarArrayHighBound*, *VarArrayRef*, *VarArrayLock*, and *VarArrayUnlock*.

When a variant containing a variant array is assigned to another variant or passed as a value parameter, the entire array is copied. Don't perform such operations unnecessarily, since they are memory-inefficient.

OleVariant

The *OleVariant* type exists on both the Windows and Linux platforms. The main difference between *Variant* and *OleVariant* is that *Variant* can contain data types that only the current application knows what to do with. *OleVariant* can only contain the data types defined as compatible with OLE Automation which means that the data types that can be passed between programs or across the network without worrying about whether the other end will know how to handle the data.

When you assign a *Variant* that contains custom data (such as a Pascal string, or a one of the new custom variant types) to an *OleVariant*, the runtime library tries to convert the *Variant* into one of the *OleVariant* standard data types (such as a Pascal string converts to an OLE BSTR string). For example, if a variant containing an *AnsiString* is assigned to an *OleVariant*, the *AnsiString* becomes a *WideString*. The same is true when passing a *Variant* to an *OleVariant* function parameter.

Type compatibility and identity

To understand which operations can be performed on which expressions, we need to distinguish several kinds of compatibility among types and values. These include *type identity*, *type compatibility*, and *assignment-compatibility*.

Type identity

Type identity is almost straightforward. When one type identifier is declared using another type identifier, without qualification, they denote the same type. Thus, given the declarations

```
type
  T1 = Integer;
  T2 = T1;
  T3 = Integer;
  T4 = T2;
```

T1, *T2*, *T3*, *T4*, and *Integer* all denote the same type. To create distinct types, repeat the word **type** in the declaration. For example,

```
type TMyInteger = type Integer;
```

creates a new type called *TMyInteger* which is not identical to *Integer*.

Language constructions that function as type names denote a different type each time they occur. Thus the declarations

```
type
  TS1 = set of Char;
  TS2 = set of Char;
```

create two distinct types, *TS1* and *TS2*. Similarly, the variable declarations

```
var
  S1: string[10];
  S2: string[10];
```

create two variables of distinct types. To create variables of the same type, use

```
var S1, S2: string[10];
```

or

```
type MyString = string[10];
var
  S1: MyString;
  S2: MyString;
```

Type compatibility

Every type is compatible with itself. Two distinct types are compatible if they satisfy at least one of the following conditions.

- They are both real types.
- They are both integer types.
- One type is a subrange of the other.
- Both types are subranges of the same type.
- Both are set types with compatible base types.
- Both are packed-string types with the same number of components.
- One is a string type and the other is a string, packed-string, or *Char* type.
- One type is *Variant* and the other is an integer, real, string, character, or Boolean type.
- Both are class, class-reference, or interface types, and one type is derived from the other.
- One type is *PChar* or *PWideChar* and the other is a zero-based character array of the form `array[0..n] of Char`.
- One type is *Pointer* (an untyped pointer) and the other is any pointer type.
- Both types are (typed) pointers to the same type and the **{$T+}** compiler directive is in effect.
- Both are procedural types with the same result type, the same number of parameters, and type-identity between parameters in corresponding positions.

Assignment-compatibility

Assignment-compatibility is not a symmetric relation. An expression of type *T2* can be assigned to a variable of type *T1* if the value of the expression falls in the range of *T1* and at least one of the following conditions is satisfied.

- *T1* and *T2* are of the same type, and it is not a file type or structured type that contains a file type at any level.
- *T1* and *T2* are compatible ordinal types.
- *T1* and *T2* are both real types.
- *T1* is a real type and *T2* is an integer type.
- *T1* is *PChar* or any string type and the expression is a string constant.
- *T1* and *T2* are both string types.
- *T1* is a string type and *T2* is a *Char* or packed-string type.
- *T1* is a long string and *T2* is *PChar*.
- *T1* and *T2* are compatible packed-string types.
- *T1* and *T2* are compatible set types.
- *T1* and *T2* are compatible pointer types.
- *T1* and *T2* are both class, class-reference, or interface types and *T2* is a derived from *T1*.
- *T1* is an interface type and *T2* is a class type that implements *T1*.
- *T1* is *PChar* or *PWideChar* and *T2* is a zero-based character array of the form `array[0..n] of Char`.

- *T1* and *T2* are compatible procedural types. (A function or procedure identifier is treated, in certain assignment statements, as an expression of a procedural type. See “Procedural types in statements and expressions” on page 5-30.)
- *T1* is *Variant* and *T2* is an integer, real, string, character, Boolean, or interface type.
- *T1* is an integer, real, string, character, or Boolean type and *T2* is *Variant*.
- *T1* is the *IUnknown* or *IDispatch* interface type and *T2* is *Variant*. (The variant’s type code must be *varEmpty*, *varUnknown*, or *varDispatch* if *T1* is *IUnknown*, and *varEmpty* or *varDispatch* if *T1* is *IDispatch*.)

Declaring types

A type declaration specifies an identifier that denotes a type. The syntax for a type declaration is

`type` *newTypeName* `=` *type*

where *newTypeName* is a valid identifier. For example, given the type declaration

```
type TMyString = string;
```

you can make the variable declaration

```
var S: TMyString;
```

A type identifier’s scope doesn’t include the type declaration itself (except for pointer types). So you cannot, for example, define a record type that uses itself recursively.

When you declare a type that is identical to an existing type, the compiler treats the new type identifier as an alias for the old one. Thus, given the declarations

```
type TValue = Real;
var
  X: Real;
  Y: TValue;
```

X and *Y* are of the same type; at runtime, there is no way to distinguish *TValue* from *Real*. This is usually of little consequence, but if your purpose in defining a new type is to utilize runtime type information—for example, to associate a property editor with properties of a particular type—the distinction between “different name” and “different type” becomes important. In this case, use the syntax

`type` *newTypeName* `= type` *type*

For example,

```
type TValue = type Real;
```

forces the compiler to create a new, distinct type called *TValue*.

Variables

A variable is an identifier whose value can change at runtime. Put differently, a variable is a name for a location in memory; you can use the name to read or write to the memory location. Variables are like containers for data, and, because they are typed, they tell the compiler how to interpret the data they hold.

Declaring variables

The basic syntax for a variable declaration is

`var` *identifierList*`:` *type*`;`

where *identifierList* is a comma-delimited list of valid identifiers and *type* is any valid type. For example,

```
var I: Integer;
```

declares a variable *I* of type *Integer*, while

```
var X, Y: Real;
```

declares two variables—*X* and *Y*—of type *Real*.

Consecutive variable declarations do not have to repeat the reserved word **var**:

```
var
  X, Y, Z: Double;
  I, J, K: Integer;
  Digit: 0..9;
  Okay: Boolean;
```

Variables declared within a procedure or function are sometimes called *local*, while other variables are called *global*. Global variables can be initialized at the same time they are declared, using the syntax

`var` *identifier*`:` *type* `=` *constantExpression*`;`

where *constantExpression* is any constant expression representing a value of type *type*. (For more information about constant expressions, see "Constant expressions" on page 5-41.) Thus the declaration

```
var I: Integer = 7;
```

is equivalent to the declaration and statement

```
var I: Integer;
 ⋮
I := 7;
```

Multiple variable declarations (such as `var X, Y, Z: Real;`) cannot include initializations, nor can declarations of variant and file-type variables.

If you don't explicitly initialize a global variable, the compiler initializes it to 0. Local variables, in contrast, cannot be initialized in their declarations and contain random data until a value is assigned to them.

When you declare a variable, you are allocating memory which is freed automatically when the variable is no longer used. In particular, local variables exist only until the program exits from the function or procedure in which they are declared. For more information about variables and memory management, see Chapter 11, "Memory management".

Absolute addresses

You can create a new variable that resides at the same address as another variable. To do so, put the directive **absolute** after the type name in the declaration of the new variable, followed by the name of an existing (previously declared) variable. For example,

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

specifies that the variable *StrLen* should start at the same address as *Str*. Since the first byte of a short string contains the string's length, the value of *StrLen* is the length of *Str*.

You cannot initialize a variable in an **absolute** declaration or combine **absolute** with any other directives.

Dynamic variables

You can create dynamic variables by calling the *GetMem* or *New* procedure. Such variables are allocated on the heap and are not managed automatically. Once you create one, it is your responsibility ultimately to free the variable's memory; use *FreeMem* to destroy variables created by *GetMem* and *Dispose* to destroy variables created by *New*. Other standard routines that operate on dynamic variables include *ReallocMem*, *Initialize*, *StrAlloc*, and *StrDispose*.

Long strings, wide strings, dynamic arrays, variants, and interfaces are also heap-allocated dynamic variables, but their memory is managed automatically.

Thread-local variables

Thread-local (or *thread*) variables are used in multithreaded applications. A thread-local variable is like a global variable, except that each thread of execution gets its own private copy of the variable, which cannot be accessed from other threads. Thread-local variables are declared with **threadvar** instead of **var**. For example,

```
threadvar X: Integer;
```

Thread-variable declarations

- cannot occur within a procedure or function.
- cannot include initializations.
- cannot specify the **absolute** directive.

Do not create pointer- or procedural-type thread variables, and do not use thread variables in dynamically loadable libraries (other than packages).

Dynamic variables that are ordinarily managed by the compiler—long strings, wide strings, dynamic arrays, variants, and interfaces—can be declared with **threadvar**, but the compiler does not automatically free the heap-allocated memory created by each thread of execution. If you use these data types in thread variables, it is your responsibility to dispose of their memory. For example,

```
threadvar S: AnsiString;
S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
```

```
 ⋮
S := '';  // free the memory used by S
```

(You can free a variant by setting it to *Unassigned* and an interface or dynamic array by setting it to **nil**.)

Declared constants

Several different language constructions are referred to as “constants”. There are numeric constants (also called *numerals*) like `17`, and string constants (also called *character strings* or *string literals*) like `'Hello world!'`; for information about numeric and string constants, see Chapter 4, “Syntactic elements”. Every enumerated type defines constants that represent the values of that type. There are predefined constants like *True*, *False*, and **nil**. Finally, there are constants that, like variables, are created individually by declaration.

Declared constants are either *true constants* or *typed constants*. These two kinds of constant are superficially similar, but they are governed by different rules and used for different purposes.

True constants

A true constant is a declared identifier whose value cannot change. For example,

```
const MaxValue = 237;
```

declares a constant called *MaxValue* that returns the integer 237. The syntax for declaring a true constant is

`const` *identifier* `=` *constantExpression*

where *identifier* is any valid identifier and *constantExpression* is an expression that the compiler can evaluate without executing your program. (See “Constant expressions” on page 5-41 for more information.)

If *constantExpression* returns an ordinal value, you can specify the type of the declared constant using a value typecast. For example

```
const MyNumber = Int64(17);
```

declares a constant called *MyNumber*, of type *Int64*, that returns the integer 17. Otherwise, the type of the declared constant is the type of the *constantExpression*.

- If *constantExpression* is a character string, the declared constant is compatible with any string type. If the character string is of length 1, it is also compatible with any character type.

- If *constantExpression* is a real, its type is *Extended*. If it is an integer, its type is given by the table below.

Table 5.8 Types for integer constants

Range of constant (hexadecimal)	Range of constant (decimal)	Type
–$8000000000000000..–$80000001	-2^{63}..–2147483649	*Int64*
–$80000000..–$8001	–2147483648..–32769	*Integer*
–$8000..–$81	–32768..–129	*Smallint*
–$80..–1	–128..–1	*Shortint*
0..$7F	0..127	0..127
$80..$FF	128..255	*Byte*
$0100..$7FFF	256..32767	0..32767
$8000..$FFFF	32768..65535	*Word*
$10000..$7FFFFFFF	65536..2147483647	0..2147483647
$80000000..$FFFFFFFF	2147483648..4294967295	*Cardinal*
$100000000..$7FFFFFFFFFFFFFFF	4294967296..$2^{63}-1$	*Int64*

Here are some examples of constant declarations:

```
const
  Min = 0;
  Max = 100;
  Center = (Max - Min) div 2;
  Beta = Chr(225);
  NumChars = Ord('Z') - Ord('A') + 1;
  Message = 'Out of memory';
  ErrStr = ' Error: ' + Message + '. ';
  ErrPos = 80 - Length(ErrStr) div 2;
  Ln10 = 2.302585092994045684;
  Ln10R = 1 / Ln10;
  Numeric = ['0'..'9'];
  Alpha = ['A'..'Z', 'a'..'z'];
  AlphaNum = Alpha + Numeric;
```

Constant expressions

A *constant expression* is an expression that the compiler can evaluate without executing the program in which it occurs. Constant expressions include numerals; character strings; true constants; values of enumerated types; the special constants *True*, *False*, and **nil**; and expressions built exclusively from these elements with operators, typecasts, and set constructors. Constant expressions cannot include variables, pointers, or function calls, except calls to the following predefined functions:

Abs
Chr
Hi
High
Length
Lo
Low
Odd
Ord
Pred
Round
SizeOf
Succ
Swap
Trunc

This definition of a *constant expression* is used in several places in Object Pascal's syntax specification. Constant expressions are required for initializing global variables, defining subrange types, assigning ordinalities to values in enumerated types, specifying default parameter values, writing **case** statements, and declaring both true and typed constants.

Examples of constant expressions:

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Borland' + ' ' + 'Developer'
Chr(32)
Ord('Z') - Ord('A') + 1
```

Resource strings

Resource strings are stored as resources and linked into the executable or library so that they can be modified without recompiling the program. For more information, see the online Help topics on localizing applications.

Resource strings are declared like other true constants, except that the word **const** is replaced by **resourcestring**. The expression to the right of the **=** symbol must be a constant expression and must return a string value. For example,

```
resourcestring
  CreateError = 'Cannot create file %s';       {  for explanations of format specifiers, }
  OpenError = 'Cannot open file %s';           { see 'Format strings' in the online Help }
  LineTooLong = 'Line too long';
  ProductName = 'Borland Rocks\000\000';
  SomeResourceString = SomeTrueConstant;
```

The compiler automatically resolves naming conflicts among resource strings in different libraries.

Typed constants

Typed constants, unlike true constants, can hold values of array, record, procedural, and pointer types. Typed constants cannot occur in constant expressions.

In the default **{$J-}** compiler state, typed constants can not have new values assigned to them; they are, in effect, read-only variables. However, if the **{$J+}** compiler directive is in effect, typed constants can have new values assigned to them; they behave essentially like initialized variables.

Declare a typed constant like this:

`const` *identifier*: *type* = *value*

where *identifier* is any valid identifier, *type* is any type except files and variants, and *value* is an expression of type *type*. For example,

```
const Max: Integer = 100;
```

In most cases, *value* must be a constant expression; but if *type* is an array, record, procedural, or pointer type, special rules apply.

Array constants

To declare an array constant, enclose the values of the array's elements, separated by commas, in parentheses at the end of the declaration. These values must be represented by constant expressions. For example,

```
const Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

declares a typed constant called *Digits* that holds an array of characters.

Zero-based character arrays often represent null-terminated strings, and for this reason string constants can be used to initialize character arrays. So the declaration above can be more conveniently represented as

```
const Digits: array[0..9] of Char = '0123456789';
```

To define a multidimensional array constant, enclose the values of each dimension in a separate set of parentheses, separated by commas. For example,

```
type TCube = array[0..1, 0..1, 0..1] of Integer;
const Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6,7)));
```

creates an array called *Maze* where

Maze[0,0,0] = 0
Maze[0,0,1] = 1
Maze[0,1,0] = 2
Maze[0,1,1] = 3
Maze[1,0,0] = 4
Maze[1,0,1] = 5
Maze[1,1,0] = 6
Maze[1,1,1] = 7

Array constants cannot contain file-type values at any level.

Record constants

To declare a record constant, specify the value of each field—as *fieldName*: *value*, with the field assignments separated by semicolons—in parentheses at the end of the declaration. The values must be represented by constant expressions. The fields must be listed in the order in which they appear in the record type declaration, and the tag field, if there is one, must have a value specified; if the record has a variant part, only the variant selected by the tag field can be assigned values.

Examples:

```
type
  TPoint = record
    X, Y: Single;
  end;
  TVector = array[0..1] of TPoint;
  TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
  TDate = record
    D: 1..31;
    M: TMonth;
    Y: 1900..1999;
  end;
const
  Origin: TPoint = (X: 0.0; Y: 0.0);
  Line: TVector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
  SomeDay: TDate = (D: 2; M: Dec; Y: 1960);
```

Record constants cannot contain file-type values at any level.

Procedural constants

To declare a procedural constant, specify the name of a function or procedure that is compatible with the declared type of the constant. For example,

```
function Calc(X, Y: Integer): Integer;
begin
 ⋮
end;

type TFunction = function(X, Y: Integer): Integer;
const MyFunction: TFunction = Calc;
```

Given these declarations, you can use the procedural constant *MyFunction* in a function call:

```
I := MyFunction(5, 7)
```

You can also assign the value **nil** to a procedural constant.

Pointer constants

When you declare a pointer constant, you must initialize it to a value that can be resolved—at least as a relative address—at compile time. There are three ways to do this: with the @ operator, with **nil**, and (if the constant is of type *PChar*) with a string literal. For example, if *I* is a global variable of type *Integer*, you can declare a constant like

```
const PI: ^Integer = @I;
```

The compiler can resolve this because global variables are part of the code segment. So are functions and global constants:

```
const PF: Pointer = @MyFunction;
```

Because string literals are allocated as global constants, you can initialize a *PChar* constant with a string literal:

```
const WarningStr: PChar = 'Warning!';
```

Addresses of local (stack-allocated) and dynamic (heap-allocated) variables cannot be assigned to pointer constants.

Chapter

6

Procedures and functions

Procedures and functions, referred to collectively as *routines*, are self-contained statement blocks that can be called from different locations in a program. A *function* is a routine that returns a value when it executes. A *procedure* is a routine that does not return a value.

Function calls, because they return a value, can be used as expressions in assignments and operations. For example,

```
I := SomeFunction(X);
```

calls *SomeFunction* and assigns the result to *I*. Function calls cannot appear on the left side of an assignment statement.

Procedure calls—and, when extended syntax is enabled (**{$X+}**), function calls—can be used as complete statements. For example,

```
DoSomething;
```

calls the *DoSomething* routine; if *DoSomething* is a function, its return value is discarded.

Procedures and functions can call themselves recursively.

Declaring procedures and functions

When you declare a procedure or function, you specify its name, the number and type of parameters it takes, and, in the case of a function, the type of its return value; this part of the declaration is sometimes called the *prototype*, *heading*, or *header*. Then you write a block of code that executes whenever the procedure or function is called; this part is sometimes called the routine's *body* or *block*.

The standard procedure *Exit* can occur within the body of any procedure or function. *Exit* halts execution of the routine where it occurs and immediately passes program control back to the point from which the routine was called.

Procedure declarations

A procedure declaration has the form

```
procedure procedureName(parameterList); directives;
localDeclarations;
begin
  statements
end;
```

where *procedureName* is any valid identifier, *statements* is a sequence of statements that execute when the procedure is called, and `(`*parameterList*`)`, *directives*`;`, and *localDeclarations*`;` are optional.

- For information about the *parameterList*, see “Parameters” on page 6-11.
- For information about *directives*, see “Calling conventions” on page 6-4, “Forward and interface declarations” on page 6-6, “External declarations” on page 6-6, “Overloading procedures and functions” on page 6-8, and “Writing dynamically loadable libraries” on page 9-3. If you include more than one directive, separate them with semicolons.
- For information about *localDeclarations*, which declares local identifiers, see “Local declarations” on page 6-10.

Here is an example of a procedure declaration:

```
procedure NumString(N: Integer; var S: string);
var
  V: Integer;
begin
  V := Abs(N);
  S := '';
  repeat
    S := Chr(V mod 10 + Ord('0')) + S;
    V := V div 10;
  until V = 0;
  if N < 0 then S := '-' + S;
end;
```

Given this declaration, you can call the *NumString* procedure like this:

```
NumString(17, MyString);
```

This procedure call assigns the value “17” to *MyString* (which must be a **string** variable).

Within a procedure’s statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the procedure. You can also use the parameter names from the parameter list (like *N* and *S* in the example above); the parameter list defines a set of local variables, so don’t try to redeclare the parameter names in the *localDeclarations* section. Finally, you can use any identifiers within whose scope the procedure declaration falls.

Function declarations

A function declaration is like a procedure declaration except that it specifies a return type and a return value. Function declarations have the form

```
function functionName(parameterList): returnType; directives;
localDeclarations;
begin
  statements
end;
```

where *functionName* is any valid identifier, *returnType* is any type, *statements* is a sequence of statements that execute when the function is called, and `(`*parameterList*`)`, *directives*`;`, and *localDeclarations*`;` are optional.

- For information about the *parameterList*, see "Parameters" on page 6-11.
- For information about *directives*, see "Calling conventions" on page 6-4, "Forward and interface declarations" on page 6-6, "External declarations" on page 6-6, "Overloading procedures and functions" on page 6-8, and "Writing dynamically loadable libraries" on page 9-3. If you include more than one directive, separate them with semicolons.
- For information about *localDeclarations*, which declares local identifiers, see "Local declarations" on page 6-10.

The function's statement block is governed by the same rules that apply to procedures. Within the statement block, you can use variables and other identifiers declared in the *localDeclarations* part of the function, parameter names from the parameter list, and any identifiers within whose scope the function declaration falls. In addition, the function name itself acts as a special variable that holds the function's return value, as does the predefined variable *Result*.

For example,

```
function WF: Integer;
begin
  WF := 17;
end;
```

defines a constant function called *WF* that takes no parameters and always returns the integer value 17. This declaration is equivalent to

```
function WF: Integer;
begin
  Result := 17;
end;
```

Here is a more complicated function declaration:

```
function Max(A: array of Real; N: Integer): Real;
var
  X: Real;
  I: Integer;
begin
  X := A[0];
```

```
    for I := 1 to N - 1 do
      if X < A[I] then X := A[I];
    Max := X;
  end;
```

You can assign a value to *Result* or to the function name repeatedly within a statement block, as long as you assign only values that match the declared return type. When execution of the function terminates, whatever value was last assigned to *Result* or to the function name becomes the function's return value. For example,

```
function Power(X: Real; Y: Integer): Real;
var
  I: Integer;
begin
  Result := 1.0;
  I := Y;
  while I > 0 do
  begin
    if Odd(I) then Result := Result * X;
    I := I div 2;
    X := Sqr(X);
  end;
end;
```

Result and the function name always represent the same value. Hence

```
function MyFunction: Integer;
begin
  MyFunction := 5;
  Result := Result * 2;
  MyFunction := Result + 1;
end;
```

returns the value 11. But *Result* is not completely interchangeable with the function name. When the function name appears on the left side of an assignment statement, the compiler assumes that it is being used (like *Result*) to track the return value; when the function name appears anywhere else in the statement block, the compiler interprets it as a recursive call to the function itself. *Result*, on the other hand, can be used as a variable in operations, typecasts, set constructors, indexes, and calls to other routines.

As long as extended syntax is enabled (**{$X+}**), *Result* is implicitly declared in every function. Do not try to redeclare it.

If the function exits without assigning a value to *Result* or the function name, then the function's return value is undefined. An exception to this rule is when the result type is a reference counted type such as a string or dynamic array. In the case of a reference-counted return type, the compiler initializes the value to 0 (an empty string or array). However, it is much safer if you always explicitly assign a return value.

Calling conventions

When you declare a procedure or function, you can specify a *calling convention* using one of the directives **register**, **pascal**, **cdecl**, **stdcall**, and **safecall**. For example,

```
function MyFunction(X, Y: Real): Real; cdecl;
 ⋮
```

Calling conventions determine the order in which parameters are passed to the routine. They also affect the removal of parameters from the stack, the use of registers for passing parameters, and error and exception handling. The default calling convention is **register**.

- The **register** and **pascal** conventions pass parameters from left to right; that is, the left most parameter is evaluated and passed first and the rightmost parameter is evaluated and passed last. The **cdecl**, **stdcall**, and **safecall** conventions pass parameters from right to left.
- For all conventions except **cdecl**, the procedure or function removes parameters from the stack upon returning. With the **cdecl** convention, the caller removes parameters from the stack when the call returns.
- The **register** convention uses up to three CPU registers to pass parameters, while the other conventions pass all parameters on the stack.
- The **safecall** convention implements exception "firewalls." On Windows, this implements interprocess COM error notification.

The table below summarizes calling conventions.

Table 6.1 Calling conventions

Directive	Parameter order	Clean-up	Passes parameters in registers?
register	Left-to-right	Routine	Yes
pascal	Left-to-right	Routine	No
cdecl	Right-to-left	Caller	No
stdcall	Right-to-left	Routine	No
safecall	Right-to-left	Routine	No

The default **register** convention is the most efficient, since it usually avoids creation of a stack frame. (Access methods for published properties must use **register**.) The **cdecl** convention is useful when you call functions from shared libraries written in C or C++, while **stdcall** and **safecall** are recommended, in general, for calls to external code. On Windows, the operating system APIs are **stdcall** and **safecall**. Other operating systems generally use **cdecl**. (Note that **stdcall** is more efficient than **cdecl**.)

The **safecall** convention must be used for declaring dual-interface methods (see Chapter 10, "Object interfaces"). The **pascal** convention is maintained for backward compatibility. For more information on calling conventions, see Chapter 12, "Program control".

The directives **near**, **far**, and **export** refer to calling conventions in 16-bit Windows programming. They have no effect in 32-bit applications and are maintained for backward compatibility only.

Forward and interface declarations

The **forward** directive replaces the block, including local variable declarations and statements, in a procedure or function declaration. For example,

```
function Calculate(X, Y: Integer): Real; forward;
```

declares a function called *Calculate*. Somewhere after the **forward** declaration, the routine must be redeclared in a *defining declaration* that includes a block. The defining declaration for *Calculate* might look like this:

```
function Calculate;
 ⋮  { declarations }
begin
 ⋮  { statement block }
end;
```

Ordinarily, a defining declaration does not have to repeat the routine's parameter list or return type, but if it does repeat them, they must match those in the **forward** declaration exactly (except that default parameters can be omitted). If the **forward** declaration specifies an overloaded procedure or function (see "Overloading procedures and functions" on page 6-8), then the defining declaration must repeat the parameter list.

A **forward** declaration and its defining declaration must appear in the same **type** declaration section. That is, you can't add a new section (such as a **var** section or **const** section) between the forward declaration and the defining declaration. The defining declaration can be an **external** or **assembler** declaration, but it cannot be another **forward** declaration.

The purpose of a **forward** declaration is to extend the scope of a procedure or function identifier to an earlier point in the source code. This allows other procedures and functions to call the **forward**-declared routine before it is actually defined. Besides letting you organize your code more flexibly, **forward** declarations are sometimes necessary for mutual recursions.

The **forward** directive has no effect in the **interface** section of a unit. Procedure and function headers in the **interface** section behave like **forward** declarations and must have defining declarations in the **implementation** section. A routine declared in the **interface** section is available from anywhere else in the unit and from any other unit or program that uses the unit where it is declared.

External declarations

The **external** directive, which replaces the block in a procedure or function declaration, allows you to call routines that are compiled separately from your program. External routines can come from object files or dynamically loadable libraries.

When importing a C++ function that takes a variable number of parameters, use the **varargs** directive. For example,

```
function printf(Format: PChar): Integer; cdecl; varargs;
```

The **varargs** directive works only with external routines and only with the **cdecl** calling convention.

Linking to object files

To call routines from a separately compiled object file, first link the object file to your application using the **$L** (or **$LINK**) compiler directive. For example,

```
On Windows: {$L BLOCK.OBJ}
```

```
On Linux:   {$L block.o}
```

links BLOCK.OBJ (Windows) or block.o (Linux) into the program or unit in which it occurs. Next, declare the functions and procedures that you want to call:

```
procedure MoveWord(var Source, Dest; Count: Integer); external;
procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
```

Now you can call the *MoveWord* and *FillWord* routines from BLOCK.OBJ (Windows) or block.o (Linux).

Declarations like the ones above are frequently used to access external routines written in assembly language. You can also place assembly-language routines directly in your Object Pascal source code; for more information, see Chapter 13, “Inline assembly code”.

Importing functions from libraries

To import routines from a dynamically loadable library (.so or .DLL), attach a directive of the form

`external` *stringConstant*`;`

to the end of a normal procedure or function header, where *stringConstant* is the name of the library file in single quotation marks. For example, on Windows

```
function SomeFunction(S: string): string; external 'strlib.dll';
```

imports a function called *SomeFunction* from strlib.dll.

On Linux,

```
function SomeFunction(S: string): string; external 'strlib.so';
```

imports a function called *SomeFunction* from strlib.so.

You can import a routine under a different name from the one it has in the library. If you do this, specify the original name in the **external** directive:

`external` *stringConstant*$_1$ `name` *stringConstant*$_2$`;`

where the first *stringConstant* gives the name of the library file and the second *stringConstant* is the routine’s original name.

On Windows: For example, the following declaration imports a function from user32.dll (part of the Windows API).

```
function MessageBox(HWnd: Integer; Text, Caption: PChar; Flags: Integer): Integer;
  stdcall; external 'user32.dll' name 'MessageBoxA';
```

The function’s original name is *MessageBoxA*, but it is imported as *MessageBox*.

Instead of a name, you can use a number to identify the routine you want to import:

```
external stringConstant index integerConstant;
```

where *integerConstant* is the routine's index in the export table.

On Linux: For example, the following declaration imports a standard system function from libc.so.6.

```
function OpenFile(const PathName: PChar; Flags: Integer): Integer; cdecl;
  external 'libc.so.6' name 'open';
```

The function's original name is *open*, but it is imported as *OpenFile*.

In your importing declaration, be sure to match the exact spelling and case of the routine's name. Later, when you call the imported routine, the name is case-insensitive.

For more information about libraries, see Chapter 9, "Libraries and packages".

Overloading procedures and functions

You can declare more than one routine in the same scope with the same name. This is called *overloading*. Overloaded routines must be declared with the **overload** directive and must have distinguishing parameter lists. For example, consider the declarations

```
function Divide(X, Y: Real): Real; overload;
begin
  Result := X/Y;
end;

function Divide(X, Y: Integer): Integer; overload;
begin
  Result := X div Y;
end;
```

These declarations create two functions, both called *Divide*, that take parameters of different types. When you call *Divide*, the compiler determines which function to invoke by looking at the actual parameters passed in the call. For example, `Divide(6.0, 3.0)` calls the first *Divide* function, because its arguments are real-valued.

You can pass to an overloaded routine parameters that are not identical in type with those in any of the routine's declarations, but that are assignment-compatible with the parameters in more than one declaration. This happens most frequently when a routine is overloaded with different integer types or different real types—for example,

```
procedure Store(X: Longint); overload;
procedure Store(X: Shortint); overload;
```

In these cases, when it is possible to do so without ambiguity, the compiler invokes the routine whose parameters are of the type with the smallest range that accommodates the actual parameters in the call. (Remember that real-valued constant expressions are always of type *Extended*.)

Overloaded routines must be distinguished by the number of parameters they take or the types of their parameters. Hence the following pair of declarations causes a compilation error.

```
function Cap(S: string): string; overload;
 ⋮
procedure Cap(var Str: string); overload;
 ⋮
```

But the declarations

```
function Func(X: Real; Y: Integer): Real; overload;
 ⋮
function Func(X: Integer; Y: Real): Real; overload;
 ⋮
```

are legal.

When an overloaded routine is declared in a **forward** or interface declaration, the defining declaration must repeat the routine's parameter list.

The compiler can distinguish between overloaded functions that contain AnsiString/PChar and WideString/WideChar parameters in the same parameter position. String constants or literals passed into such an overload situation are translated into the native string or character type, which is AnsiString/PChar.

```
procedure test(const S: String);  overload;
procedure test(const W: WideString); overload;

var
    a: string;
    b: widestring;
begin
   a := 'a';
   b := 'b';
   test(a);    // calls String version
   test(b);    // calls WideString version
   test('abc');    // calls String version
   test(WideString('abc'));   // calls widestring version
end;
```

Variants can also be used as parameters in overloaded function declarations. Variant is considered more general than any simple type. Preference is always given to exact type matches over variant matches. If a variant is passed into such an overload situation, and an overload that takes a variant exists in that parameter position, it is considered to be an exact match for the Variant type.

This can cause some minor side effects with float types. Float types are matched by size. If there is no exact match for the float variable passed to the overload call but a variant parameter is available, the variant is taken over any smaller float type.

For example:

```
procedure foo(i: integer); overload;
procedure foo(d: double);  overload;
procedure foo(v: variant); overload;
var
  v: variant;
```

```
begin
  foo(1);       // integer version
  foo(v);       // variant version
  foo(1.2);     // variant version (float literals -> extended precision)
end;
```

This example calls the variant version of foo, not the double version, because the 1.2 constant is implicitly an extended type and extended is not an exact match for double. Extended is also not an exact match for variant, but variant is considered a more general type (whereas double is a smaller type than extended).

```
foo(Double(1.2));
```

This typecast does not work. You should use typed consts instead.

```
const  d: double = 1.2;
  begin
    foo(d);
  end;
```

The above code works correctly, and calls the double version.

```
const  s: single = 1.2;
begin
  foo(s);
end;
```

The above code also calls the double version of foo. Single is a better fit to double than to variant.

When declaring a set of overloaded routines, the best way to avoid float promotion to variant is to declare a version of your overloaded function for each float type (Single, Double, Extended) along with the variant version.

If you use default parameters in overloaded routines, be careful of ambiguous parameter signatures. For more information, see "Default parameters and overloaded routines" on page 6-19.

You can limit the potential effects of overloading by qualifying a routine's name when you call it. For example, `Unit1.MyProcedure(X, Y)` can call only routines declared in *Unit1*; if no routine in *Unit1* matches the name and parameter list in the call, an error results.

For information about distributing overloaded methods in a class hierarchy, see "Overloading methods" on page 7-12. For information about exporting overloaded routines from a shared library, see "The exports clause" on page 9-5.

Local declarations

The body of a function or procedure often begins with declarations of local variables used in the routine's statement block. These declarations can also include constants, types, and other routines. The scope of a local identifier is limited to the routine where it is declared.

Nested routines

Functions and procedures sometimes contain other functions and procedures within the local-declarations section of their blocks. For example, the following declaration of a procedure called *DoSomething* contains a nested procedure.

```
procedure DoSomething(S: string);
var
  X, Y: Integer;

  procedure NestedProc(S: string);
  begin
    ⋮
  end;

begin
   ⋮
  NestedProc(S);
   ⋮
end;
```

The scope of a nested routine is limited to the procedure or function in which it is declared. In our example, *NestedProc* can be called only within *DoSomething*.

For real examples of nested routines, look at the *DateTimeToString* procedure, the *ScanDate* function, and other routines in the *SysUtils* unit.

Parameters

Most procedure and function headers include a *parameter list*. For example, in the header

```
function Power(X: Real; Y: Integer): Real;
```

the parameter list is `(X: Real; Y: Integer)`.

A parameter list is a sequence of parameter declarations separated by semicolons and enclosed in parentheses. Each declaration is a comma-delimited series of parameter names, followed in most cases by a colon and a type identifier, and in some cases by the = symbol and a default value. Parameter names must be valid identifiers. Any declaration can be preceded by one of the reserved words **var**, **const**, and **out**. Examples:

```
(X, Y: Real)
(var S: string; X: Integer)
(HWnd: Integer; Text, Caption: PChar; Flags: Integer)
(const P; I: Integer)
```

The parameter list specifies the number, order, and type of parameters that must be passed to the routine when it is called. If a routine does not take any parameters, omit the identifier list and the parentheses in its declaration:

```
procedure UpdateRecords;
begin
 ⋮
end;
```

Within the procedure or function body, the parameter names (*X* and *Y* in the first example above) can be used as local variables. Do not redeclare the parameter names in the local declarations section of the procedure or function body.

Parameter semantics

Parameters are categorized in several ways:

- Every parameter is classified as *value*, *variable*, *constant*, or *out*. Value parameters are the default; the reserved words **var**, **const**, and **out** indicate variable, constant, and out parameters, respectively.
- Value parameters are always *typed*, while constant, variable, and out parameters can be either typed or *untyped*.
- Special rules apply to array parameters. See "Array parameters" on page 6-15.

Files and instances of structured types that contain files can be passed only as variable (**var**) parameters.

Value and variable parameters

Most parameters are either value parameters (the default) or variable (**var**) parameters. Value parameters are passed *by value*, while variable parameters are passed *by reference*. To see what this means, consider the following functions.

```
function DoubleByValue(X: Integer): Integer;     // X is a value parameter
begin
  X := X * 2;
  Result := X;
end;

function DoubleByRef(var X: Integer): Integer;   // X is a variable parameter
begin
  X := X * 2;
  Result := X;
end;
```

These functions return the same result, but only the second one—*DoubleByRef*—can change the value of a variable passed to it. Suppose we call the functions like this:

```
var
  I, J, V, W: Integer;
begin
  I := 4;
  V := 4;
  J := DoubleByValue(I);   // J = 8, I = 4
  W := DoubleByRef(V);     // W = 8, V = 8
end;
```

After this code executes, the variable *I*, which was passed to *DoubleByValue*, has the same value we initially assigned to it. But the variable *V*, which was passed to *DoubleByRef*, has a different value.

A value parameter acts like a local variable that gets initialized to the value passed in the procedure or function call. If you pass a variable as a value parameter, the procedure or function creates a copy of it; changes made to the copy have no effect on the original variable and are lost when program execution returns to the caller.

A variable parameter, on the other hand, acts like a pointer rather than a copy. Changes made to the parameter within the body of a function or procedure persist after program execution returns to the caller and the parameter name itself has gone out of scope.

Even if the same variable is passed in two or more **var** parameters, no copies are made. This is illustrated in the following example.

```
procedure AddOne(var X, Y: Integer);
begin
  X := X + 1;
  Y := Y + 1;
end;

var I: Integer;
begin
  I := 1;
  AddOne(I, I);
end;
```

After this code executes, the value of *I* is 3.

If a routine's declaration specifies a **var** parameter, you must pass an assignable expression—that is, a variable, typed constant (in the **{$J+}** state), dereferenced pointer, field, or indexed variable—to the routine when you call it. To use our previous examples, `DoubleByRef(7)` produces an error, although `DoubleByValue(7)` is legal.

Indexes and pointer dereferences passed in **var** parameters—for example, `DoubleByRef(MyArray[I])`—are evaluated once, before execution of the routine.

Constant parameters

A constant (**const**) parameter is like a local constant or read-only variable. Constant parameters are similar to value parameters, except that you can't assign a value to a constant parameter within the body of a procedure or function, nor can you pass one as a **var** parameter to another routine. (But when you pass an object reference as a constant parameter, you can still modify the object's properties.)

Using **const** allows the compiler to optimize code for structured- and string-type parameters. It also provides a safeguard against unintentionally passing a parameter by reference to another routine.

Here, for example, is the header for the *CompareStr* function in the *SysUtils* unit:

```
function CompareStr(const S1, S2: string): Integer;
```

Because *S1* and *S2* are not modified in the body of *CompareStr*, they can be declared as constant parameters.

Out parameters

An **out** parameter, like a variable parameter, is passed by reference. With an **out** parameter, however, the initial value of the referenced variable is discarded by the routine it is passed to. The **out** parameter is for output only; that is, it tells the function or procedure where to store output, but doesn't provide any input.

For example, consider the procedure heading

```
procedure GetInfo(out Info: SomeRecordType);
```

When you call *GetInfo*, you must pass it a variable of type *SomeRecordType*:

```
var MyRecord: SomeRecordType;
 ⋮
GetInfo(MyRecord);
```

But you're not using *MyRecord* to pass any data to the *GetInfo* procedure; *MyRecord* is just a container where you want *GetInfo* to store the information it generates. The call to *GetInfo* immediately frees the memory used by *MyRecord*, before program control passes to the procedure.

Out parameters are frequently used with distributed-object models like COM and CORBA. In addition, you should use **out** parameters when you pass an uninitialized variable to a function or procedure.

Untyped parameters

You can omit type specifications when declaring **var**, **const**, and **out** parameters. (Value parameters must be typed.) For example,

```
procedure TakeAnything(const C);
```

declares a procedure called *TakeAnything* that accepts a parameter of any type. When you call such a routine, you cannot pass it a numeral or untyped numeric constant.

Within a procedure or function body, untyped parameters are incompatible with every type. To operate on an untyped parameter, you must cast it. In general, the compiler cannot verify that operations on untyped parameters are valid.

The following example uses untyped parameters in a function called *Equal* that compares a specified number of bytes of any two variables.

```
function Equal(var Source, Dest; Size: Integer): Boolean;
type
  TBytes = array[0..MaxInt - 1] of Byte;
var
  N: Integer;
begin
  N := 0;
  while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do
    Inc(N);
  Equal := N = Size;
end;
```

Given the declarations

```
type
  TVector = array[1..10] of Integer;
  TPoint = record
    X, Y: Integer;
  end;
```

```
var
  Vec1, Vec2: TVector;
  N: Integer;
  P: TPoint;
```

you could make the following calls to *Equal*:

```
Equal(Vec1, Vec2, SizeOf(TVector))           // compare Vec1 to Vec2
Equal(Vec1, Vec2, SizeOf(Integer) * N)       // compare first N elements of Vec1 and Vec2
Equal(Vec1[1], Vec1[6], SizeOf(Integer) * 5)  // compare first 5 to last 5 elements of Vec1
Equal(Vec1[1], P, 4)                         // compare Vec1[1] to P.X and Vec1[2] to P.Y
```

String parameters

When you declare routines that take short-string parameters, you cannot include length specifiers in the parameter declarations. That is, the declaration

```
procedure Check(S: string[20]);   // syntax error
```

causes a compilation error. But

```
type TString20 = string[20];
procedure Check(S: TString20);
```

is valid. The special identifier *OpenString* can be used to declare routines that take short-string parameters of varying length:

```
procedure Check(S: OpenString);
```

When the **{$H–}** and **{$P+}** compiler directives are both in effect, the reserved word **string** is equivalent to *OpenString* in parameter declarations.

Short strings, *OpenString*, **$H**, and **$P** are supported for backward compatibility only. In new code, you can avoid these considerations by using long strings.

Array parameters

When you declare routines that take array parameters, you cannot include index type specifiers in the parameter declarations. That is, the declaration

```
procedure Sort(A: array[1..10] of Integer);   // syntax error
```

causes a compilation error. But

```
type TDigits = array[1..10] of Integer;
procedure Sort(A: TDigits);
```

is valid. For most purposes, however, *open array parameters* are a better solution.

Open array parameters

Open array parameters allow arrays of different sizes to be passed to the same procedure or function. To define a routine with an open array parameter, use the syntax `array of` *type* (rather than `array[`*X*`..`*Y*`] of` *type*) in the parameter declaration. For example,

```
function Find(A: array of Char): Integer;
```

declares a function called *Find* that takes a character array of any size and returns an integer.

Note The syntax of open array parameters resembles that of dynamic array types, but they do not mean the same thing. The example above creates a function that takes any array of *Char* elements, including (but not limited to) dynamic arrays. To declare parameters that must be dynamic arrays, you need to specify a type identifier:

```
type TDynamicCharArray = array of Char;
function Find(A: TDynamicCharArray): Integer;
```

For information about dynamic arrays, see "Dynamic arrays" on page 5-19.

Within the body of a routine, open array parameters are governed by the following rules.

- They are always zero-based. The first element is 0, the second element is 1, and so forth. The standard *Low* and *High* functions return 0 and *Length*−1, respectively. The *SizeOf* function returns the size of the actual array passed to the routine.
- They can be accessed by element only. Assignments to an entire open array parameter are not allowed.
- They can be passed to other procedures and functions only as open array parameters or untyped **var** parameters. They cannot be passed to *SetLength*.
- Instead of an array, you can pass a variable of the open array parameter's base type. It will be treated as an array of length 1.

When you pass an array as an open array value parameter, the compiler creates a local copy of the array within the routine's stack frame. Be careful not to overflow the stack by passing large arrays.

The following examples use open array parameters to define a *Clear* procedure that assigns zero to each element in an array of reals and a *Sum* function that computes the sum of the elements in an array of reals.

```
procedure Clear(var A: array of Real);
var
  I: Integer;
begin
  for I := 0 to High(A) do A[I] := 0;
end;

function Sum(const A: array of Real): Real;
var
  I: Integer;
  S: Real;
begin
  S := 0;
  for I := 0 to High(A) do S := S + A[I];
  Sum := S;
end;
```

When you call routines that use open array parameters, you can pass *open array constructors* to them. See "Open array constructors" on page 6-20.

Variant open array parameters

Variant open array parameters allow you to pass an array of differently typed expressions to a single procedure or function. To define a routine with a variant open array parameter, specify **array of const** as the parameter's type. Thus

```
procedure DoSomething(A: array of const);
```

declares a procedure called *DoSomething* that can operate on heterogeneous arrays.

The **array of const** construction is equivalent to **array of** *TVarRec*. *TVarRec*, declared in the *System* unit, represents a record with a variant part that can hold values of integer, Boolean, character, real, string, pointer, class, class reference, interface, and variant types. *TVarRec*'s *VType* field indicates the type of each element in the array. Some types are passed as pointers rather than values; in particular, long strings are passed as *Pointer* and must be typecast to **string**. See the online Help on *TVarRec* for details.

The following example uses a variant open array parameter in a function that creates a string representation of each element passed to it and concatenates the results into a single string. The string-handling routines called in this function are defined in *SysUtils*.

```
function MakeStr(const Args: array of const): string;
const
  BoolChars: array[Boolean] of Char = ('F', 'T');
var
  I: Integer;
begin
  Result := '';
  for I := 0 to High(Args) do
    with Args[I] do
      case VType of
        vtInteger:    Result := Result + IntToStr(VInteger);
        vtBoolean:    Result := Result + BoolChars[VBoolean];
        vtChar:       Result := Result + VChar;
        vtExtended:   Result := Result + FloatToStr(VExtended^);
        vtString:     Result := Result + VString^;
        vtPChar:      Result := Result + VPChar;
        vtObject:     Result := Result + VObject.ClassName;
        vtClass:      Result := Result + VClass.ClassName;
        vtAnsiString: Result := Result + string(VAnsiString);
        vtCurrency:   Result := Result + CurrToStr(VCurrency^);
        vtVariant:    Result := Result + string(VVariant^);
        vtInt64:      Result := Result + IntToStr(VInt64^);
    end;
end;
```

We can call this function using an open array constructor (see "Open array constructors" on page 6-20). For example,

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

returns the string "test100 T3.14159TForm".

Default parameters

You can specify default parameter values in a procedure or function heading. Default values are allowed only for typed **const** and value parameters. To provide a default value, end the parameter declaration with the **=** symbol followed by a constant expression that is assignment-compatible with the parameter's type.

For example, given the declaration

```
procedure FillArray(A: array of Integer; Value: Integer = 0);
```

the following procedure calls are equivalent.

```
FillArray(MyArray);
FillArray(MyArray, 0);
```

A multiple-parameter declaration cannot specify a default value. Thus, while

```
function MyFunction(X: Real = 3.5; Y: Real = 3.5): Real;
```

is legal,

```
function MyFunction(X, Y: Real = 3.5): Real;   // syntax error
```

is not.

Parameters with default values must occur at the end of the parameter list. That is, all parameters following the first declared default value must also have default values. So the following declaration is illegal.

```
procedure MyProcedure(I: Integer = 1; S: string);   // syntax error
```

Default values specified in a procedural type override those specified in an actual routine. Thus, given the declarations

```
type TResizer = function(X: Real; Y: Real = 1.0): Real;
function Resizer(X: Real; Y: Real = 2.0): Real;
var
  F: TResizer;
  N: Real;
```

the statements

```
F := Resizer;
F(N);
```

result in the values (*N*, 1.0) being passed to *Resizer*.

Default parameters are limited to values that can be specified by a constant expression. (See "Constant expressions" on page 5-41.) Hence parameters of a dynamic-array, procedural, class, class-reference, or interface type can have no value other than **nil** as their default. Parameters of a record, variant, file, static-array, or object type cannot have default values at all.

For information about calling routines with default parameter values, see "Calling procedures and functions" on page 6-19.

Default parameters and overloaded routines

If you use default parameter values in an overloaded routine, avoid ambiguous parameter signatures. Consider, for example, the following.

```
procedure Confused(I: Integer); overload;
 ⋮
procedure Confused(I: Integer; J: Integer = 0); overload;
 ⋮
Confused(X);    //  Which procedure is called?
```

In fact, neither procedure is called. This code generates a compilation error.

Default parameters in forward and interface declarations

If a routine has a **forward** declaration or appears in the interface section of a unit, default parameter values—if there are any—must be specified in the **forward** or interface declaration. In this case, the default values can be omitted from the defining (implementation) declaration; but if the defining declaration includes default values, they must match those in the **forward** or interface declaration exactly.

Calling procedures and functions

When you call a procedure or function, program control passes from the point where the call is made to the body of the routine. You can make the call using the routine's declared name (with or without qualifiers) or using a procedural variable that points to the routine. In either case, if the routine is declared with parameters, your call to it must pass parameters that correspond in order and type to the routine's parameter list. The parameters you pass to a routine are called *actual parameters*, while the parameters in the routine's declaration are called *formal parameters*.

When calling a routine, remember that

- expressions used to pass typed **const** and value parameters must be assignment-compatible with the corresponding formal parameters.
- expressions used to pass **var** and **out** parameters must be identically typed with the corresponding formal parameters, unless the formal parameters are untyped.
- only assignable expressions can be used to pass **var** and **out** parameters.
- if a routine's formal parameters are untyped, numerals and true constants with numeric values cannot be used as actual parameters.

When you call a routine that uses default parameter values, all actual parameters following the first accepted default must also use the default values; calls of the form `SomeFunction(,,X)` are not legal.

You can omit parentheses when passing all and only the default parameters to a routine. For example, given the procedure

```
procedure DoSomething(X: Real = 1.0; I: Integer = 0; S: string = '');
```

the following calls are equivalent.

```
DoSomething();
```

```
DoSomething;
```

Open array constructors

Open array constructors allow you to construct arrays directly within function and procedure calls. They can be passed only as open array parameters or variant open array parameters.

An open array constructor, like a set constructor, is a sequence of expressions separated by commas and enclosed in brackets.

For example, given the declarations

```
var I, J: Integer;
procedure Add(A: array of Integer);
```

you could call the *Add* procedure with the statement

```
Add([5, 7, I, I + J]);
```

This is equivalent to

```
var Temp: array[0..3] of Integer;
 ⋮
Temp[0] := 5;
Temp[1] := 7;
Temp[2] := I;
Temp[3] := I + J;
Add(Temp);
```

Open array constructors can be passed only as value or **const** parameters. The expressions in a constructor must be assignment-compatible with the base type of the array parameter. In the case of a variant open array parameter, the expressions can be of different types.

Chapter 7

Classes and objects

A *class*, or *class type*, defines a structure consisting of *fields*, *methods*, and *properties*. Instances of a class type are called *objects*. The fields, methods, and properties of a class are called its *components* or *members*.

- A field is essentially a variable that is part of an object. Like the fields of a record, a class's fields represent data items that exist in each instance of the class.
- A method is a procedure or function associated with a class. Most methods operate on objects—that is, instances of a class. Some methods (called *class methods*) operate on class types themselves.
- A property is an interface to data associated with an object (often stored in a field). Properties have *access specifiers*, which determine how their data is read and modified. From other parts of a program—outside of the object itself—a property appears in most respects like a field.

Objects are dynamically allocated blocks of memory whose structure is determined by their class type. Each object has a unique copy of every field defined in the class, but all instances of a class share the same methods. Objects are created and destroyed by special methods called *constructors* and *destructors*.

A variable of a class type is actually a pointer that references an object. Hence more than one variable can refer to the same object. Like other pointers, class-type variables can hold the value **nil**. But you don't have to explicitly dereference a class-type variable to access the object it points to. For example, `SomeObject.Size := 100` assigns the value 100 to the *Size* property of the object referenced by *SomeObject*; you would *not* write this as `SomeObject^.Size := 100`.

Class types

A class type must be declared and given a name before it can be instantiated. (You cannot define a class type within a variable declaration.) Declare classes only in the outermost scope of a program or unit, not in a procedure or function declaration.

A class type declaration has the form

```
type className = class (ancestorClass)
  memberList
 end;
```

where *className* is any valid identifier, `(`*ancestorClass*`)` is optional, and *memberList* declares members—that is, fields, methods, and properties—of the class. If you omit `(`*ancestorClass*`)`, then the new class inherits directly from the predefined *TObject* class. If you include `(`*ancestorClass*`)` and *memberList* is empty, you can omit `end`. A class type declaration can also include a list of *interfaces* implemented by the class; see "Implementing interfaces" on page 10-4.

Methods appear in a class declaration as function or procedure headings, with no body. Defining declarations for each method occur elsewhere in the program.

For example, here is the declaration of the *TMemoryStream* class from the *Classes* unit.

```
type
TMemoryStream = class(TCustomMemoryStream)
  private
    FCapacity: Longint;
    procedure SetCapacity(NewCapacity: Longint);
  protected
    function Realloc(var NewCapacity: Longint): Pointer; virtual;
    property Capacity: Longint read FCapacity write SetCapacity;
  public
    destructor Destroy; override;
    procedure Clear;
    procedure LoadFromStream(Stream: TStream);
    procedure LoadFromFile(const FileName: string);
    procedure SetSize(NewSize: Longint); override;
    function Write(const Buffer; Count: Longint): Longint; override;
  end;
```

TMemoryStream descends from *TStream* (in the *Classes* unit), inheriting most of its members. But it defines—or redefines—several methods and properties, including its destructor method, *Destroy*. Its constructor, *Create*, is inherited without change from *TObject*, and so is not redeclared. Each member is declared as *private*, *protected*, or *public* (this class has no *published* members); for explanations of these terms, see "Visibility of class members" on page 7-4.

Given this declaration, you can create an instance of *TMemoryStream* as follows:

```
var stream: TMemoryStream;
stream := TMemoryStream.Create;
```

Inheritance and scope

When you declare a class, you can specify its immediate ancestor. For example,

```
type TSomeControl = class(TControl);
```

declares a class called *TSomeControl* that descends from *TControl*. A class type automatically inherits all of the members from its immediate ancestor. Each class can declare new members and can redefine inherited ones, but a class cannot remove members defined in an ancestor. Hence *TSomeControl* contains all of the members defined in *TControl* and in each of *TControl*'s ancestors.

The scope of a member's identifier starts at the point where the member is declared, continues to the end of the class declaration, and extends over all descendants of the class and the blocks of all methods defined in the class and its descendants.

TObject and TClass

The *TObject* class, declared in the *System* unit, is the ultimate ancestor of all other classes. *TObject* defines only a handful of methods, including a basic constructor and destructor. In addition to *TObject*, the *System* unit declares the class-reference type *TClass*:

```
TClass = class of TObject;
```

For more information about *TObject*, see the online help. For more information about class-reference types, see "Class references" on page 7-23.

If the declaration of a class type doesn't specify an ancestor, the class inherits directly from *TObject*. Thus

```
type TMyClass = class
  ⋮
end;
```

is equivalent to

```
type TMyClass = class(TObject)
  ⋮
end;
```

The latter form is recommended for readability.

Compatibility of class types

A class type is assignment-compatible with its ancestors. Hence a variable of a class type can reference an instance of any descendant type. For example, given the declarations

```
type
  TFigure = class(TObject);
  TRectangle = class(TFigure);
  TSquare = class(TRectangle);
var
  Fig: TFigure;
```

the variable *Fig* can be assigned values of type *TFigure*, *TRectangle*, and *TSquare*.

Object types

As an alternative to class types, you can declare *object types* using the syntax

```
type objectTypeName = object (ancestorObjectType)
  memberList
end;
```

where *objectTypeName* is any valid identifier, (*ancestorObjectType*) is optional, and *memberList* declares fields, methods, and properties. If (*ancestorObjectType*) is omitted, then the new type has no ancestor. Object types cannot have published members.

Since object types do not descend from *TObject*, they provide no built-in constructors, destructors, or other methods. You can create instances of an object type using the *New* procedure and destroy them with the *Dispose* procedure, or you can simply declare variables of an object type, just as you would with records.

Object types are supported for backward compatibility only. Their use is not recommended.

Visibility of class members

Every member of a class has an attribute called *visibility*, which is indicated by one of the reserved words **private**, **protected**, **public**, **published**, or **automated**. For example,

```
published property Color: TColor read GetColor write SetColor;
```

declares a published property called *Color*. Visibility determines where and how a member can be accessed, with private representing the least accessibility, protected representing an intermediate level of accessibility, and public, published, and automated representing the greatest accessibility.

If a member's declaration appears without its own visibility specifier, the member has the same visibility as the one that precedes it. Members at the beginning of a class declaration that don't have a specified visibility are by default published, provided the class is compiled in the **{$M+}** state or is derived from a class compiled in the **{$M+}** state; otherwise, such members are public.

For readability, it is best to organize a class declaration by visibility, placing all the private members together, followed by all the protected members, and so forth. This way each visibility reserved word appears at most once and marks the beginning of a new "section" of the declaration. So a typical class declaration should like this:

```
type
  TMyClass = class(TControl)
  private
   ⋮ { private declarations here}
  protected
   ⋮ { protected declarations here }
  public
   ⋮ { public declarations here }
  published
   ⋮ { published declarations here }
  end;
```

You can increase the visibility of a member in a descendant class by redeclaring it, but you cannot decrease its visibility. For example, a protected property can be made public in a descendant, but not private. Moreover, published members cannot become public in a descendant class. For more information, see "Property overrides and redeclarations" on page 7-22.

Private, protected, and public members

A *private* member is invisible outside of the unit or program where its class is declared. In other words, a private method cannot be called from another module, and a private field or property cannot be read or written to from another module. By placing related class declarations in the same module, you can give the classes access to one another's private members without making those members more widely accessible.

A *protected* member is visible anywhere in the module where its class is declared and from any descendant class, regardless of the module where the descendant class appears. In other words, a protected method can be called, and a protected field or property read or written to, from the definition of any method belonging to a class that descends from the one where the protected member is declared. Members that are intended for use only in the implementation of derived classes are usually protected.

A *public* member is visible wherever its class can be referenced.

Published members

Published members have the same visibility as public members. The difference is that *runtime type information* (RTTI) is generated for published members. RTTI allows an application to query the fields and properties of an object dynamically and to locate its methods. RTTI is used to access the values of properties when saving and loading form files, to display properties in the Object Inspector, and to associate specific methods (called *event handlers*) with specific properties (called *events*).

Published properties are restricted to certain data types. Ordinal, string, class, interface, and method-pointer types can be published. So can set types, provided the upper and lower bounds of the base type have ordinal values between 0 and 31. (In other words, the set must fit in a byte, word, or double word.) Any real type except *Real48* can be published. Properties of an array type (as distinct from *array properties*, discussed below) cannot be published.

Some properties, although publishable, are not fully supported by the streaming system. These include properties of record types, array properties of all publishable types (see "Array properties" on page 7-19), and properties of enumerated types that include anonymous values (see "Enumerated types with explicitly assigned ordinality" on page 5-7). If you publish a property of this kind, the Object Inspector won't display it correctly, nor will the property's value be preserved when objects are streamed to disk.

All methods are publishable, but a class cannot publish two or more overloaded methods with the same name. Fields can be published only if they are of a class or interface type.

A class cannot have published members unless it is compiled in the **{$M+}** state or descends from a class compiled in the **{$M+}** state. Most classes with published members derive from *TPersistent*, which is compiled in the **{$M+}** state, so it is seldom necessary to use the **$M** directive.

Automated members

Automated members have the same visibility as public members. The difference is that *Automation type information* (required for Automation servers) is generated for automated members. Automated members typically appear only in Windows classes and is not recommended for Linux programming. The **automated** reserved word is maintained for backward compatibility. The *TAutoObject* class in the *ComObj* unit does not use **automated**.

The following restrictions apply to methods and properties declared as automated.

- The types of all properties, array property parameters, method parameters, and function results must be automatable. The automatable types are *Byte*, *Currency*, *Real*, *Double*, *Longint*, *Integer*, *Single*, *Smallint*, *AnsiString*, *WideString*, *TDateTime*, *Variant*, *OleVariant*, *WordBool*, and all interface types.
- Method declarations must use the default **register** calling convention. They can be virtual, but not dynamic.
- Property declarations can include access specifiers (**read** and **write**) but other specifiers (**index**, **stored**, **default**, and **nodefault**) are not allowed. Access specifiers must list a method identifier that uses the default **register** calling convention; field identifiers are not allowed.
- Property declarations must specify a type. Property overrides are not allowed.

The declaration of an automated method or property can include a **dispid** directive. Specifying an already used ID in a **dispid** directive causes an error.

On Windows, this directive must be followed by an integer constant that specifies an Automation dispatch ID for the member. Otherwise, the compiler automatically assigns the member a dispatch ID that is one larger than the largest dispatch ID used by any method or property in the class and its ancestors. For more information about Automation (on Windows only), see “Automation objects (Windows only)” on page 10-10.

Forward declarations and mutually dependent classes

If the declaration of a class type ends with the word **class** and a semicolon—that is, if it has the form

```
type className = class;
```

with no ancestor or class members listed after the word **class**—then it is a *forward declaration*. A forward declaration must be resolved by a *defining declaration* of the same class within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent classes. For example,

```
type
  TFigure = class;  // forward declaration
  TDrawing = class
    Figure: TFigure;
     ⋮
  end;

  TFigure = class  // defining declaration
    Drawing: TDrawing;
     ⋮
  end;
```

Do not confuse forward declarations with complete declarations of types that derive from *TObject* without declaring any class members.

```
type
  TFirstClass = class;           // this is a forward declaration

  TSecondClass = class           // this is a complete class declaration
    end;                         //

  TThirdClass = class(TObject);  // this is a complete class declaration
```

Fields

A field is like a variable that belongs to an object. Fields can be of any type, including class types. (That is, fields can hold object references.) Fields are usually private.

To define a field member of a class, simply declare the field as you would a variable. All field declarations must occur before any property or method declarations. For example, the following declaration creates a class called *TNumber* whose only member, other than the methods is inherits from *TObject*, is an integer field called *Int*.

```
type TNumber = class
  Int: Integer;
end;
```

Fields are statically bound; that is, references to them are fixed at compile time. To see what this means, consider the following code.

```
type
  TAncestor = class
    Value: Integer;
  end;

  TDescendant = class(TAncestor)
    Value: string;  // hides the inherited Value field
  end;

var
  MyObject: TAncestor;

begin
  MyObject := TDescendant.Create;
  MyObject.Value := 'Hello!';  // error
  TDescendant(MyObject).Value := 'Hello!';  // works!
end;
```

Although *MyObject* holds an instance of *TDescendant*, it is declared as *TAncestor*. The compiler therefore interprets *MyObject.Value* as referring to the (integer) field declared in *TAncestor*. Both fields, however, exist in the *TDescendant* object; the inherited *Value* is hidden by the new one, and can be accessed through a typecast.

Methods

A method is a procedure or function associated with a class. A call to a method specifies the object (or, if it is a class method, the class) that the method should operate on. For example,

```
SomeObject.Free
```

calls the *Free* method in *SomeObject*.

Method declarations and implementations

Within a class declaration, methods appear as procedure and function headings, which work like **forward** declarations. Somewhere after the class declaration, but within the same module, each method must be implemented by a defining declaration. For example, suppose the declaration of *TMyClass* includes a method called *DoSomething*:

```
type
  TMyClass = class(TObject)
     ⋮
    procedure DoSomething;
     ⋮
  end;
```

A defining declaration for *DoSomething* must occur later in the module:

```
procedure TMyClass.DoSomething;
begin
 ⋮
end;
```

While a class can be declared in either the interface or the implementation section of a unit, defining declarations for a class's methods must be in the implementation section.

In the heading of a defining declaration, the method name is always qualified with the name of the class to which it belongs. The heading can repeat the parameter list from the class declaration; if it does so, the order, type, and names of the parameters must match exactly, and, if the method is a function, so must the return value.

Method declarations can include special directives that are not used with other functions or procedures. Directives should appear in the class declaration only, not in the defining declaration, and should always be listed in the following order:

reintroduce; **overload**; *binding*; *calling convention*; **abstract**; *warning* where *binding* is **virtual**, **dynamic**, or **override**; *calling convention* is **register**, **pascal**, **cdecl**, **stdcall**, or **safecall**; and *warning* is **platform**, **deprecated**, or **library**.

Inherited

The reserved word **inherited** plays a special role in implementing polymorphic behavior. It can occur in method definitions, with or without an identifier after it.

If **inherited** is followed by the name of a member, it represents a normal method call or reference to a property or field—except that the search for the referenced member begins with the immediate ancestor of the enclosing method's class. For example, when

```
inherited Create(...);
```

occurs in the definition of a method, it calls the inherited *Create*.

When **inherited** has no identifier after it, it refers to the inherited method with the same name as the enclosing method. In this case, **inherited** takes no explicit parameters, but passes to the inherited method the same parameters with which the enclosing method was called. For example,

```
inherited;
```

occurs frequently in the implementation of constructors. It calls the inherited constructor with the same parameters that were passed to the descendant.

Self

Within the implementation of a method, the identifier *Self* references the object in which the method is called. For example, here is the implementation of *TCollection*'s *Add* method in the *Classes* unit.

```
function TCollection.Add: TCollectionItem;
begin
  Result := FItemClass.Create(Self);
end;
```

The *Add* method calls the *Create* method in the class referenced by the *FItemClass* field, which is always a *TCollectionItem* descendant. *TCollectionItem.Create* takes a single parameter of type *TCollection*, so *Add* passes it the *TCollection* instance object where *Add* is called. This is illustrated in the following code.

```
var MyCollection: TCollection;
 ⋮
MyCollection.Add  // MyCollection is passed to the TCollectionItem.Create method
```

Self is useful for a variety of reasons. For example, a member identifier declared in a class type might be redeclared in the block of one of the class's methods. In this case, you can access the original member identifier as `Self`.*Identifier*.

For information about *Self* in class methods, see "Class methods" on page 7-25.

Method binding

Methods can be *static* (the default), *virtual*, or *dynamic*. Virtual and dynamic methods can be *overridden*, and they can be *abstract*. These designations come into play when a variable of one class type holds a value of a descendant class type. They determine which implementation is activated when a method is called.

Static methods

Methods are by default static. When a static method is called, the declared (compile-time) type of the class or object variable used in the method call determines which implementation to activate. In the following example, the *Draw* methods are static.

```
type
  TFigure = class
    procedure Draw;
  end;
  TRectangle = class(TFigure)
    procedure Draw;
  end;
```

Given these declarations, the following code illustrates the effect of calling a static method. In the second call to *Figure.Draw*, the *Figure* variable references an object of class *TRectangle*, but the call invokes the implementation of *Draw* in *TFigure*, because the declared type of the *Figure* variable is *TFigure*.

```
var
  Figure: TFigure;
  Rectangle: TRectangle;
begin
  Figure := TFigure.Create;
  Figure.Draw;  // calls TFigure.Draw
  Figure.Destroy;
  Figure := TRectangle.Create;
  Figure.Draw;  // calls TFigure.Draw
  TRectangle(Figure).Draw;  // calls TRectangle.Draw
  Figure.Destroy;
  Rectangle := TRectangle.Create;
  Rectangle.Draw;  // calls TRectangle.Draw
  Rectangle.Destroy;
end;
```

Virtual and dynamic methods

To make a method virtual or dynamic, include the **virtual** or **dynamic** directive in its declaration. Virtual and dynamic methods, unlike static methods, can be *overridden* in descendant classes. When an overridden method is called, the actual (runtime) type of the class or object used in the method call—not the declared type of the variable—determines which implementation to activate.

To override a method, redeclare it with the **override** directive. An **override** declaration must match the ancestor declaration in the order and type of its parameters and in its result type (if any).

In the following example, the *Draw* method declared in *TFigure* is overridden in two descendant classes.

```
type
  TFigure = class
    procedure Draw; virtual;
  end;
  TRectangle = class(TFigure)
    procedure Draw; override;
  end;
  TEllipse = class(TFigure)
    procedure Draw; override;
  end;
```

Given these declarations, the following code illustrates the effect of calling a virtual method through a variable whose actual type varies at runtime.

```
var
  Figure: TFigure;
begin
  Figure := TRectangle.Create;
  Figure.Draw;  // calls TRectangle.Draw
  Figure.Destroy;
  Figure := TEllipse.Create;
  Figure.Draw;  // calls TEllipse.Draw
  Figure.Destroy;
end;
```

Only virtual and dynamic methods can be overridden. All methods, however, can be *overloaded*; see “Overloading methods”.

Virtual versus dynamic

Virtual and dynamic methods are semantically equivalent. They differ only in the implementation of method-call dispatching at runtime. Virtual methods optimize for speed, while dynamic methods optimize for code size.

In general, virtual methods are the most efficient way to implement polymorphic behavior. Dynamic methods are useful when a base class declares many overridable methods which are inherited by many descendant classes in an application, but only occasionally overridden.

Overriding versus hiding

If a method declaration specifies the same method identifier and parameter signature as an inherited method, but doesn’t include **override**, the new declaration merely *hides* the inherited one without overriding it. Both methods exist in the descendant class, where the method name is statically bound. For example,

```
type
  T1 = class(TObject)
    procedure Act; virtual;
  end;
  T2 = class(T1)
    procedure Act;  // Act is redeclared, but not overridden
  end;
```

```
var
  SomeObject: T1;
begin
  SomeObject := T2.Create;
  SomeObject.Act;  // calls T1.Act
end;
```

Reintroduce

The **reintroduce** directive suppresses compiler warnings about hiding previously declared virtual methods. For example,

```
procedure DoSomething; reintroduce;  // the ancestor class also has a DoSomething method
```

Use **reintroduce** when you want to hide an inherited virtual method with a new one.

Abstract methods

An abstract method is a virtual or dynamic method that has no implementation in the class where it is declared. Its implementation is deferred to a descendant class. Abstract methods must be declared with the directive **abstract** after **virtual** or **dynamic**. For example,

```
procedure DoSomething; virtual; abstract;
```

You can call an abstract method only in a class or instance of a class in which the method has been overridden.

Overloading methods

A method can be redeclared using the **overload** directive. In this case, if the redeclared method has a different parameter signature from its ancestor, it overloads the inherited method without hiding it. Calling the method in a descendant class activates whichever implementation matches the parameters in the call.

If you overload a virtual method, use the **reintroduce** directive when you redeclare it in descendant classes. For example,

```
type
  T1 = class(TObject)
    procedure Test(I: Integer); overload; virtual;
  end;
  T2 = class(T1)
    procedure Test(S: string); reintroduce; overload;
  end;
   ⋮
  SomeObject := T2.Create;
  SomeObject.Test('Hello!');  // calls T2.Test
  SomeObject.Test(7);         // calls T1.Test
```

Within a class, you cannot publish multiple overloaded methods with the same name. Maintenance of runtime type information requires a unique name for each published member.

```
type
  TSomeClass = class
  published
    function Func(P: Integer): Integer;
    function Func(P: Boolean): Integer  // error
     ⋮
```

Methods that serve as property **read** or **write** specifiers cannot be overloaded.

The implementation of an overloaded method must repeat the parameter list from the class declaration. For more information about overloading, see “Overloading procedures and functions” on page 6-8.

Constructors

A constructor is a special method that creates and initializes instance objects. The declaration of a constructor looks like a procedure declaration, but it begins with the word **constructor**. Examples:

```
constructor Create;
constructor Create(AOwner: TComponent);
```

Constructors must use the default **register** calling convention. Although the declaration specifies no return value, a constructor returns a reference to the object it creates or is called in.

A class can have more than one constructor, but most have only one. It is conventional to call the constructor *Create*.

To create an object, call the constructor method in a class type. For example,

```
MyObject := TMyClass.Create;
```

This allocates storage for the new object on the heap, sets the values of all ordinal fields to zero, assigns **nil** to all pointer and class-type fields, and makes all string fields empty. Other actions specified in the constructor implementation are performed next; typically, objects are initialized based on values passed as parameters to the constructor. Finally, the constructor returns a reference to the newly allocated and initialized object. The type of the returned value is the same as the class type specified in the constructor call.

If an exception is raised during execution of a constructor that was invoked on a class reference, the *Destroy* destructor is automatically called to destroy the unfinished object.

When a constructor is called using an object reference (rather than a class reference), it does not create an object. Instead, the constructor operates on the specified object, executing only the statements in the constructor’s implementation, and then returns a reference to the object. A constructor is typically invoked on an object reference in conjunction with the reserved word **inherited** to execute an inherited constructor.

Here is an example of a class type and its constructor.

```
type
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    procedure PenChanged(Sender: TObject);
    procedure BrushChanged(Sender: TObject);
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    ⋮
  end;

constructor TShape.Create(Owner: TComponent);
begin
  inherited Create(Owner);  // Initialize inherited parts
  Width := 65;  // Change inherited properties
  Height := 65;
  FPen := TPen.Create;  // Initialize new fields
  FPen.OnChange := PenChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := BrushChanged;
end;
```

The first action of a constructor is usually to call an inherited constructor to initialize the object's inherited fields. The constructor then initializes the fields introduced in the descendant class. Because a constructor always clears the storage it allocates for a new object, all fields start with a value of zero (ordinal types), **nil** (pointer and class types), empty (string types), or *Unassigned* (variants). Hence there is no need to initialize fields in a constructor's implementation except to nonzero or nonempty values.

When invoked through a class-type identifier, a constructor declared as **virtual** is equivalent to a static constructor. When combined with class-reference types, however, virtual constructors allow polymorphic construction of objects—that is, construction of objects whose types aren't known at compile time. (See "Class references" on page 7-23.)

Destructors

A destructor is a special method that destroys the object where it is called and deallocates its memory. The declaration of a destructor looks like a procedure declaration, but it begins with the word **destructor**. Examples:

```
destructor Destroy;
destructor Destroy; override;
```

Destructors must use the default **register** calling convention. Although a class can have more than one destructor, it is recommended that each class override the inherited *Destroy* method and declare no other destructors.

To call a destructor, you must reference an instance object. For example,

```
MyObject.Destroy;
```

When a destructor is called, actions specified in the destructor implementation are performed first. Typically, these consist of destroying any embedded objects and freeing resources that were allocated by the object. Then the storage that was allocated for the object is disposed of.

Here is an example of a destructor implementation.

```
destructor TShape.Destroy;
begin
  FBrush.Free;
  FPen.Free;
  inherited Destroy;
end;
```

The last action in a destructor's implementation is typically to call the inherited destructor to destroy the object's inherited fields.

When an exception is raised during creation of an object, *Destroy* is automatically called to dispose of the unfinished object. This means that *Destroy* must be prepared to dispose of partially constructed objects. Because a constructor sets the fields of a new object to zero or empty values before performing other actions, class-type and pointer-type fields in a partially constructed object are always **nil**. A destructor should therefore check for **nil** values before operating on class-type or pointer-type fields. Calling the *Free* method (defined in *TObject*), rather than *Destroy*, offers a convenient way of checking for **nil** values before destroying an object.

Message methods

Message methods implement responses to dynamically dispatched messages. The message method syntax is supported on all platforms. The VCL uses message methods to respond to Windows messages. CLX does not use message methods to respond to system events.

A message method is created by including the **message** directive in a method declaration, followed by an integer constant between 1 and 49151 which specifies the message ID. For message methods in VCL controls, the integer constant can be one of the Windows message IDs defined, along with corresponding record types, in the *Messages* unit. A message method must be a procedure that takes a single **var** parameter.

For example, on Windows:

```
type
  TTextBox = class(TCustomControl)
  private
    procedure WMChar(var Message: TWMChar); message WM_CHAR;
    ⋮
  end;
```

For example, on Linux or for cross-platform programming, you would handle messages as follows:

```
const
  ID_REFRESH = $0001;

type
  TTextBox = class(TCustomControl)
  private
    procedure Refresh(var Message: TMessageRecordType); message ID_REFRESH;
    ⋮
  end;
```

A message method does not have to include the **override** directive to override an inherited message method. In fact, it doesn't have to specify the same method name or parameter type as the method it overrides. The message ID alone determines which message the method responds to and whether it is an override.

Implementing message methods

The implementation of a message method can call the inherited message method, as in this example (for Windows):

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
  if Chr(Message.CharCode) = #13 then
    ProcessEnter
  else
    inherited;
end;
```

On Linux or for cross-platform programming, you would write the same example as follows:

```
procedure TTextBox.Refresh(var Message: TMessageRecordType);
begin
  if Chr(Message.Code) = #13 then
    ...
  else
    inherited;
end;
```

The **inherited** statement searches backward through the class hierarchy and invokes the first message method with the same ID as the current method, automatically passing the message record to it. If no ancestor class implements a message method for the given ID, **inherited** calls the *DefaultHandler* method originally defined in *TObject*.

The implementation of *DefaultHandler* in *TObject* simply returns without performing any actions. By overriding *DefaultHandler*, a class can implement its own default handling of messages. On Windows, the *DefaultHandler* method for VCL controls calls the Windows *DefWindowProc* function.

Message dispatching

Message handlers are seldom called directly. Instead, messages are dispatched to an object using the *Dispatch* method inherited from *TObject*:

```
procedure Dispatch(var Message);
```

The *Message* parameter passed to *Dispatch* must be a record whose first entry is a field of type *Cardinal* containing a message ID.

Dispatch searches backward through the class hierarchy (starting from the class of the object where it is called) and invokes the first message method for the ID passed to it. If no message method is found for the given ID, *Dispatch* calls *DefaultHandler*.

Properties

A property, like a field, defines an attribute of an object. But while a field is merely a storage location whose contents can be examined and changed, a property associates specific actions with reading or modifying its data. Properties provide control over access to an object's attributes, and they allow attributes to be computed.

The declaration of a property specifies a name and a type, and includes at least one access specifier. The syntax of a property declaration is

`property` *propertyName*`[`*indexes*`]:` *type* `index` *integerConstant* *specifiers*`;`

where

- *propertyName* is any valid identifier.
- `[`*indexes*`]` is optional and is a sequence of parameter declarations separated by semicolons. Each parameter declaration has the form *identifier*$_1$, ..., *identifier*$_n$: *type*. For more information, see "Array properties" on page 7-19.
- *type* must be a predefined or previously declared data type. That is, property declarations like `property Num: 0..9 ...` are invalid.
- the `index` *integerConstant* clause is optional. For more information, see "Index specifiers" on page 7-20.
- *specifiers* is a sequence of **read**, **write**, **stored**, **default** (or **nodefault**), and **implements** specifiers. Every property declaration must have at least one **read** or **write** specifier. (For information about **implements**, see "Implementing interfaces by delegation" on page 10-6.)

Properties are defined by their access specifiers. Unlike fields, properties cannot be passed as **var** parameters, nor can the @ operator be applied to a property. The reason is that a property doesn't necessarily exist in memory. It could, for instance, have a **read** method that retrieves a value from a database or generates a random value.

Property access

Every property has a **read** specifier, a **write** specifier, or both. These are called *access specifiers* and they have the form

`read` *fieldOrMethod*
`write` *fieldOrMethod*

where *fieldOrMethod* is the name of a field or method declared in the same class as the property or in an ancestor class.

- If *fieldOrMethod* is declared in the same class, it must occur before the property declaration. If it is declared in an ancestor class, it must be visible from the descendant; that is, it cannot be a private field or method of an ancestor class declared in a different unit.
- If *fieldOrMethod* is a field, it must be of the same type as the property.
- If *fieldOrMethod* is a method, it cannot be **dynamic** and, if **virtual**, cannot be overloaded. Moreover, access methods for a published property must use the default **register** calling convention.
- In a **read** specifier, if *fieldOrMethod* is a method, it must be a parameterless function whose result type is the same as the property's type. (An exception is the access method for an indexed property. See "Array properties" on page 7-19.)
- In a **write** specifier, if *fieldOrMethod* is a method, it must be a procedure that takes a single value or **const** parameter of the same type as the property (or two, if it is an array property).

For example, given the declaration

```
property Color: TColor read GetColor write SetColor;
```

the *GetColor* method must be declared as

```
function GetColor: TColor;
```

and the *SetColor* method must be declared as one of these:

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: TColor);
```

(The name of *SetColor*'s parameter, of course, doesn't have to be *Value*.)

When a property is referenced in an expression, its value is read using the field or method listed in the **read** specifier. When a property is referenced in an assignment statement, its value is written using the field or method listed in the **write** specifier.

The example below declares a class called *TCompass* with a published property called *Heading*. The value of *Heading* is read through the *FHeading* field and written through the *SetHeading* procedure.

```
type
  THeading = 0..359;
  TCompass = class(TControl)
  private
    FHeading: THeading;
    procedure SetHeading(Value: THeading);
  published
    property Heading: THeading read FHeading write SetHeading;
    ⋮
  end;
```

Given this declaration, the statements

```
if Compass.Heading = 180 then GoingSouth;
```

```
Compass.Heading := 135;
```

correspond to

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

In the *TCompass* class, no action is associated with reading the *Heading* property; the **read** operation consists of retrieving the value stored in the *FHeading* field. On the other hand, assigning a value to the *Heading* property translates into a call to the *SetHeading* method, which, presumably, stores the new value in the *FHeading* field as well as performing other actions. For example, *SetHeading* might be implemented like this:

```
procedure TCompass.SetHeading(Value: THeading);
begin
  if FHeading <> Value then
  begin
    FHeading := Value;
    Repaint;  // update user interface to reflect new value
  end;
end;
```

A property whose declaration includes only a **read** specifier is a *read-only* property, and one whose declaration includes only a **write** specifier is a *write-only* property. It is an error to assign a value to a read-only property or use a write-only property in an expression.

Array properties

Array properties are indexed properties. They can represent things like items in a list, child controls of a control, and pixels of a bitmap.

The declaration of an array property includes a parameter list that specifies the names and types of the indexes. For example,

```
property Objects[Index: Integer]: TObject read GetObject write SetObject;
property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
property Values[const Name: string]: string read GetValue write SetValue;
```

The format of an index parameter list is the same as that of a procedure's or function's parameter list, except that the parameter declarations are enclosed in brackets instead of parentheses. Unlike arrays, which can use only ordinal-type indexes, array properties allow indexes of any type.

For array properties, access specifiers must list methods rather than fields. The method in a **read** specifier must be a function that takes the number and type of parameters listed in the property's index parameter list, in the same order, and whose result type is identical to the property's type. The method in a **write** specifier must be a procedure that takes the number and type of parameters listed in the property's index parameter list, in the same order, plus an additional value or **const** parameter of the same type as the property.

For example, the access methods for the array properties above might be declared as

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

An array property is accessed by indexing the property identifier. For example, the statements

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\DELPHI\BIN';
```

correspond to

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\DELPHI\BIN');
```

On Linux, you would use a path such as '/usr/local/bin' in place of 'C:\DELPHI\BIN' in the above example.

The definition of an array property can be followed by the **default** directive, in which case the array property becomes the default property of the class. For example,

```
type
  TStringArray = class
  public
    property Strings[Index: Integer]: string ...; default;
    ⋮
  end;
```

If a class has a default property, you can access that property with the abbreviation *object*`[`*index*`]`, which is equivalent to *object*`.`*property*`[`*index*`]`. For example, given the declaration above, `StringArray.Strings[7]` can be abbreviated to `StringArray[7]`. A class can have only one default property. Changing or hiding the default property in descendant classes may lead to unexpected behavior, since the compiler always determines an object's default property statically.

Index specifiers

Index specifiers allow several properties to share the same access method while representing different values. An index specifier consists of the directive **index** followed by an integer constant between –2147483647 and 2147483647. If a property has an index specifier, its **read** and **write** specifiers must list methods rather than fields. For example,

```
type
  TRectangle = class
  private
    FCoordinates: array[0..3] of Longint;
    function GetCoordinate(Index: Integer): Longint;
    procedure SetCoordinate(Index: Integer; Value: Longint);
  public
```

```
    property Left: Longint index 0 read GetCoordinate write SetCoordinate;
    property Top: Longint index 1 read GetCoordinate write SetCoordinate;
    property Right: Longint index 2 read GetCoordinate write SetCoordinate;
    property Bottom: Longint index 3 read GetCoordinate write SetCoordinate;
    property Coordinates[Index: Integer]: Longint read GetCoordinate write SetCoordinate;
    ⋮
  end;
```

An access method for a property with an index specifier must take an extra value parameter of type *Integer*. For a **read** function, it must be the last parameter; for a **write** procedure, it must be the second-to-last parameter (preceding the parameter that specifies the property value). When a program accesses the property, the property's integer constant is automatically passed to the access method.

Given the declaration above, if *Rectangle* is of type *TRectangle*, then

```
Rectangle.Right := Rectangle.Left + 100;
```

corresponds to

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

Storage specifiers

The optional **stored**, **default**, and **nodefault** directives are called *storage specifiers*. They have no effect on program behavior, but control the way runtime type information (RTTI) is maintained. Specifically, storage specifiers determine whether or not to save the values of published properties in form files.

The **stored** directive must be followed by *True*, *False*, the name of a *Boolean* field, or the name of a parameterless method that returns a *Boolean* value. For example,

```
property Name: TComponentName read FName write SetName stored False;
```

If a property has no **stored** directive, it is treated as if `stored True` were specified.

The **default** directive must be followed by a constant of the same type as the property. For example,

```
property Tag: Longint read FTag write FTag default 0;
```

To override an inherited **default** value without specifying a new one, use the **nodefault** directive. The **default** and **nodefault** directives are supported only for ordinal types and for set types, provided the upper and lower bounds of the set's base type have ordinal values between 0 and 31; if such a property is declared without **default** or **nodefault**, it is treated as if **nodefault** were specified. For reals, pointers, and strings, there is an implicit **default** value of 0, **nil**, and `''` (the empty string), respectively.

When saving a component's state, the storage specifiers of the component's published properties are checked. If a property's current value is different from its **default** value (or if there is no **default** value) and the **stored** specifier is *True*, then the property's value is saved. Otherwise, the property's value is not saved.

Note Storage specifiers are not supported for array properties. The **default** directive has a different meaning when used in an array property declaration. See "Array properties" on page 7-19.

Property overrides and redeclarations

A property declaration that doesn't specify a type is called a *property override*. Property overrides allow you to change a property's inherited visibility or specifiers. The simplest override consists only of the reserved word **property** followed by an inherited property identifier; this form is used to change a property's visibility. For example, if an ancestor class declares a property as protected, a derived class can redeclare it in a public or published section of the class. Property overrides can include **read**, **write**, **stored**, **default**, and **nodefault** directives; any such directive overrides the corresponding inherited directive. An override can replace an inherited access specifier, add a missing specifier, or increase a property's visibility, but it cannot remove an access specifier or decrease a property's visibility. An override can include an **implements** directive, which adds to the list of implemented interfaces without removing inherited ones.

The following declarations illustrate the use of property overrides.

```
type
  TAncestor = class
   ⋮
  protected
    property Size: Integer read FSize;
    property Text: string read GetText write SetText;
    property Color: TColor read FColor write SetColor stored False;
    ⋮
  end;
type
  TDerived = class(TAncestor)
   ⋮
  protected
    property Size write SetSize;
  published
    property Text;
    property Color stored True default clBlue;
    ⋮
  end;
```

The override of *Size* adds a **write** specifier to allow the property to be modified. The overrides of *Text* and *Color* change the visibility of the properties from protected to published. The property override of *Color* also specifies that the property should be filed if its value isn't *clBlue*.

A redeclaration of a property that includes a type identifier hides the inherited property rather than overriding it. This means that a new property is created with the same name as the inherited one. Any property declaration that specifies a type must be a complete declaration, and must therefore include at least one access specifier.

Whether a property is hidden or overridden in a derived class, property look-up is always *static*. That is, the declared (compile-time) type of the variable used to identify an object determines the interpretation of its property identifiers. Hence, after the following code executes, reading or assigning a value to *MyObject.Value* invokes *Method1* or *Method2*, even though *MyObject* holds an instance of *TDescendant*. But you can cast *MyObject* to *TDescendant* to access the descendant class's properties and their access specifiers.

```
type
  TAncestor = class
     ⋮
    property Value: Integer read Method1 write Method2;
  end;

  TDescendant = class(TAncestor)
     ⋮
    property Value: Integer read Method3 write Method4;
  end;

var MyObject: TAncestor;
 ⋮
MyObject := TDescendant.Create;
```

Class references

Sometimes operations are performed on a class itself, rather than on instances of a class (that is, objects). This happens, for example, when you call a constructor method using a class reference. You can always refer to a specific class using its name, but at times it is necessary to declare variables or parameters that take classes as values, and in these situations you need *class-reference types*.

Class-reference types

A class-reference type, sometimes called a *metaclass*, is denoted by a construction of the form

`class of` *type*

where *type* is any class type. The identifier *type* itself denotes a value whose type is `class of` *type*. If $type_1$ is an ancestor of $type_2$, then `class of` $type_2$ is assignment-compatible with `class of` $type_1$. Thus

```
type TClass = class of TObject;
var AnyObj: TClass;
```

declares a variable called *AnyObj* that can hold a reference to any class. (The definition of a class-reference type cannot occur directly in a variable declaration or parameter list.) You can assign the value **nil** to a variable of any class-reference type.

To see how class-reference types are used, look at the declaration of the constructor for *TCollection* (in the *Classes* unit):

```
type TCollectionItemClass = class of TCollectionItem;
  ⋮
constructor Create(ItemClass: TCollectionItemClass);
```

This declaration says that to create a *TCollection* instance object, you must pass to the constructor the name of a class descending from *TCollectionItem*.

Class-reference types are useful when you want to invoke a class method or virtual constructor on a class or object whose actual type is unknown at compile time.

Constructors and class references

A constructor can be called using a variable of a class-reference type. This allows construction of objects whose type isn't known at compile time. For example,

```
type TControlClass = class of TControl;

function CreateControl(ControlClass: TControlClass;
  const ControlName: string; X, Y, W, H: Integer): TControl;
begin
  Result := ControlClass.Create(MainForm);
  with Result do
  begin
    Parent := MainForm;
    Name := ControlName;
    SetBounds(X, Y, W, H);
    Visible := True;
  end;
end;
```

The *CreateControl* function requires a class-reference parameter to tell it what kind of control to create. It uses this parameter to call the class's constructor. Because class-type identifiers denote class-reference values, a call to *CreateControl* can specify the identifier of the class to create an instance of. For example,

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

Constructors called using class references are usually virtual. The constructor implementation activated by the call depends on the runtime type of the class reference.

Class operators

Every class inherits from *TObject* methods called *ClassType* and *ClassParent* that return, respectively, a reference to the class of an object and of an object's immediate ancestor. Both methods return a value of type *TClass* (where `TClass = class of TObject`), which can be cast to a more specific type. Every class also inherits a method called *InheritsFrom* that tests whether the object where it is called descends from a specified class. These methods are used by the **is** and **as** operators, and it is seldom necessary to call them directly.

The is operator

The **is** operator, which performs dynamic type checking, is used to verify the actual runtime class of an object. The expression

object `is` *class*

returns *True* if *object* is an instance of the class denoted by *class* or one of its descendants, and *False* otherwise. (If *object* is **nil**, the result is *False*.) If the declared type of *object* is unrelated to *class*—that is, if the types are distinct and one is not an ancestor of the other—a compilation error results. For example,

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

This statement casts a variable to *TEdit* after first verifying that the object it references is an instance of *TEdit* or one of its descendants.

The as operator

The **as** operator performs checked typecasts. The expression

object `as` *class*

returns a reference to the same object as *object*, but with the type given by *class*. At runtime, *object* must be an instance of the class denoted by *class* or one of its descendants, or be **nil**; otherwise an exception is raised. If the declared type of *object* is unrelated to *class*—that is, if the types are distinct and one is not an ancestor of the other—a compilation error results. For example,

```
with Sender as TButton do
begin
  Caption := '&Ok';
  OnClick := OkClick;
end;
```

The rules of operator precedence often require **as** typecasts to be enclosed in parentheses. For example,

```
(Sender as TButton).Caption := '&Ok';
```

Class methods

A class method is a method (other than a constructor) that operates on classes instead of objects. The definition of a class method must begin with the reserved word **class**. For example,

```
type
  TFigure = class
  public
    class function Supports(Operation: string): Boolean; virtual;
    class procedure GetInfo(var Info: TFigureInfo); virtual;
    ⋮
  end;
```

The defining declaration of a class method must also begin with **class**. For example,

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
  ⋮
end;
```

In the defining declaration of a class method, the identifier *Self* represents the class where the method is called (which could be a descendant of the class in which it is defined). If the method is called in the class *C*, then *Self* is of the type `class of` *C*. Thus you cannot use *Self* to access fields, properties, and normal (object) methods, but you can use it to call constructors and other class methods.

A class method can be called through a class reference or an object reference. When it is called through an object reference, the class of the object becomes the value of *Self*.

Exceptions

An *exception* is raised when an error or other event interrupts normal execution of a program. The exception transfers control to an *exception handler*, which allows you to separate normal program logic from error-handling. Because exceptions are objects, they can be grouped into hierarchies using inheritance, and new exceptions can be introduced without affecting existing code. An exception can carry information, such as an error message, from the point where it is raised to the point where it is handled.

When an application uses the *SysUtils* unit, all runtime errors are automatically converted into exceptions. Errors that would otherwise terminate an application—such as insufficient memory, division by zero, and general protection faults—can be caught and handled.

When to use exceptions

Exceptions provide an elegant way to trap runtime errors without halting the program and without awkward conditional statements. The complexity of Object Pascal's exception-handling mechanism, however, makes it inefficient, and it should therefore be used judiciously. While it is possible to raise exceptions for almost any reason, and to protect almost any block of code by wrapping it in a **try...except** or **try...finally** statement, in practice these tools are best reserved for special situations.

Exception handling is appropriate for errors whose chances of occurring are low or difficult to assess, but whose consequences are likely to be catastrophic (such as crashing the application); for error conditions that are complicated or difficult to test for in **if...then** statements; and when you need to respond to exceptions raised by the operating system or by routines whose source code you don't control. Exceptions are commonly used for hardware, memory, I/O, and operating-system errors.

Conditional statements are often the best way to test for errors. For example, suppose you want to make sure that a file exists before trying to open it. You could do it this way:

```
try
  AssignFile(F, FileName);
  Reset(F);  // raises an EInOutError exception if file is not found
except
  on Exception do ...
end;
```

But you could also avoid the overhead of exception handling by using

```
if FileExists(FileName) then  // returns False if file is not found; raises no exception
```

```
begin
  AssignFile(F, FileName);
  Reset(F);
end;
```

Assertions provide another way of testing a Boolean condition anywhere in your source code. When an *Assert* statement fails, the program either halts or (if it uses the *SysUtils* unit) raises an *EAssertionFailed* exception. Assertions should be used only to test for conditions that you do not expect to occur. For more information, see the online Help for the standard procedure *Assert*.

Declaring exception types

Exception types are declared just like other classes. In fact, it is possible to use an instance of any class as an exception, but it is recommended that exceptions be derived from the *Exception* class defined in *SysUtils*.

You can group exceptions into families using inheritance. For example, the following declarations in *SysUtils* define a family of exception types for math errors.

```
type
  EMathError = class(Exception);
  EInvalidOp = class(EMathError);
  EZeroDivide = class(EMathError);
  EOverflow = class(EMathError);
  EUnderflow = class(EMathError);
```

Given these declarations, you can define a single *EMathError* exception handler that also handles *EInvalidOp*, *EZeroDivide*, *EOverflow*, and *EUnderflow*.

Exception classes sometimes define fields, methods, or properties that convey additional information about the error. For example,

```
type EInOutError = class(Exception)
    ErrorCode: Integer;
  end;
```

Raising and handling exceptions

To create an exception object, call the exception class's constructor within a **raise** statement. For example,

```
raise EMathError.Create;
```

In general, the form of a **raise** statement is

`raise` *object* `at` *address*

where *object* and `at` *address* are both optional. If *object* is omitted, the statement re-raises the current exception; see "Re-raising exceptions" on page 7-30. When an *address* is specified, it is usually a pointer to a procedure or function; use this option to raise the exception from an earlier point in the stack than the one where the error actually occurred.

When an exception is *raised*—that is, referenced in a **raise** statement—it is governed by special exception-handling logic. A **raise** statement never returns control in the normal way. Instead, it transfers control to the innermost exception handler that can handle exceptions of the given class. (The innermost handler is the one whose **try...except** block was most recently entered but has not yet exited.)

For example, the function below converts a string to an integer, raising an *ERangeError* exception if the resulting value is outside a specified range.

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
  Result := StrToInt(S);  // StrToInt is declared in SysUtils
  if (Result < Min) or (Result > Max) then
    raise ERangeError.CreateFmt(
      '%d is not within the valid range of %d..%d',
      [Result, Min, Max]);
end;
```

Notice the *CreateFmt* method called in the **raise** statement. *Exception* and its descendants have special constructors that provide alternative ways to create exception messages and context IDs. See the online Help for details.

A raised exception is destroyed automatically after it is handled. Never attempt to destroy a raised exception manually.

Note Raising an exception in the initialization section of a unit may not produce the intended result. Normal exception support comes from the *SysUtils* unit, which must be initialized before such support is available. If an exception occurs during initialization, all initialized units—including *SysUtils*—are finalized and the exception is re-raised. Then the exception is caught and handled, usually by interrupting the program.

Try...except statements

Exceptions are handled within **try...except** statements. For example,

```
try
  X := Y/Z;
except
  on EZeroDivide do HandleZeroDivide;
end;
```

This statement attempts to divide *Y* by *Z*, but calls a routine named *HandleZeroDivide* if an *EZeroDivide* exception is raised.

The syntax of a **try...except** statement is

`try` *statements* `except` *exceptionBlock* `end`

where *statements* is a sequence of statements (delimited by semicolons) and *exceptionBlock* is either

- another sequence of statements or
- a sequence of exception handlers, optionally followed by

  `else` *statements*

An exception handler has the form

`on` *identifier*`:` *type* `do` *statement*

where *identifier*`:` is optional (if included, *identifier* can be any valid identifier), *type* is a type used to represent exceptions, and *statement* is any statement.

A **try...except** statement executes the statements in the initial *statements* list. If no exceptions are raised, the exception block (*exceptionBlock*) is ignored and control passes to the next part of the program.

If an exception is raised during execution of the initial *statements* list, either by a **raise** statement in the *statements* list or by a procedure or function called from the *statements* list, an attempt is made to "handle" the exception:

- If any of the handlers in the exception block matches the exception, control passes to the first such handler. An exception handler "matches" an exception just in case the *type* in the handler is the class of the exception or an ancestor of that class.
- If no such handler is found, control passes to the *statement* in the **else** clause, if there is one.
- If the exception block is just a sequence of statements without any exception handlers, control passes to the first statement in the list.

If none of the conditions above is satisfied, the search continues in the exception block of the next-most-recently entered **try...except** statement that has not yet exited. If no appropriate handler, **else** clause, or statement list is found there, the search propagates to the next-most-recently entered **try...except** statement, and so forth. If the outermost **try...except** statement is reached and the exception is still not handled, the program terminates.

When an exception is handled, the stack is traced back to the procedure or function containing the **try...except** statement where the handling occurs, and control is transferred to the executed exception handler, **else** clause, or statement list. This process discards all procedure and function calls that occurred after entering the **try...except** statement where the exception is handled. The exception object is then automatically destroyed through a call to its *Destroy* destructor and control is passed to the statement following the **try...except** statement. (If a call to the *Exit*, *Break*, or *Continue* standard procedure causes control to leave the exception handler, the exception object is still automatically destroyed.)

In the example below, the first exception handler handles division-by-zero exceptions, the second one handles overflow exceptions, and the final one handles all other math exceptions. *EMathError* appears last in the exception block because it is the ancestor of the other two exception classes; if it appeared first, the other two handlers would never be invoked.

```
try
 ⋮
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
end;
```

An exception handler can specify an identifier before the name of the exception class. This declares the identifier to represent the exception object during execution of the statement that follows **on...do**. The scope of the identifier is limited to that statement. For example,

```
try
 ⋮
except
  on E: Exception do ErrorDialog(E.Message, E.HelpContext);
end;
```

If the exception block specifies an **else** clause, the **else** clause handles any exceptions that aren't handled by the block's exception handlers. For example,

```
try
 ⋮
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
  on EMathError do HandleMathError;
else
  HandleAllOthers;
end;
```

Here, the **else** clause handles any exception that isn't an *EMathError*.

An exception block that contains no exception handlers, but instead consists only of a list of statements, handles all exceptions. For example,

```
try
 ⋮
except
  HandleException;
end;
```

Here, the *HandleException* routine handles any exception that occurs as a result of executing the statements between **try** and **except**.

Re-raising exceptions

When the reserved word **raise** occurs in an exception block without an object reference following it, it raises whatever exception is handled by the block. This allows an exception handler to respond to an error in a limited way and then *re-raise* the exception. Re-raising is useful when a procedure or function has to clean up after an exception occurs but cannot fully handle the exception.

For example, the *GetFileList* function allocates a *TStringList* object and fills it with file names matching a specified search path:

```
function GetFileList(const Path: string): TStringList;
var
  I: Integer;
  SearchRec: TSearchRec;
begin
  Result := TStringList.Create;
  try
    I := FindFirst(Path, 0, SearchRec);
```

```
      while I = 0 do
      begin
        Result.Add(SearchRec.Name);
        I := FindNext(SearchRec);
      end;
    except
      Result.Free;
      raise;
    end;
  end;
```

GetFileList creates a *TStringList* object, then uses the *FindFirst* and *FindNext* functions (defined in *SysUtils*) to initialize it. If the initialization fails—for example because the search path is invalid, or because there is not enough memory to fill in the string list—*GetFileList* needs to dispose of the new string list, since the caller does not yet know of its existence. For this reason, initialization of the string list is performed in a **try...except** statement. If an exception occurs, the statement's exception block disposes of the string list, then re-raises the exception.

Nested exceptions

Code executed in an exception handler can itself raise and handle exceptions. As long as these exceptions are also handled within the exception handler, they do not affect the original exception. However, once an exception raised in an exception handler propagates beyond that handler, the original exception is lost. This is illustrated by the *Tan* function below.

```
type
  ETrigError = class(EMathError);

function Tan(X: Extended): Extended;
begin
  try
    Result := Sin(X) / Cos(X);
  except
    on EMathError do
      raise ETrigError.Create('Invalid argument to Tan');
  end;
end;
```

If an *EMathError* exception occurs during execution of *Tan*, the exception handler raises an *ETrigError*. Since *Tan* does not provide a handler for *ETrigError*, the exception propagates beyond the original exception handler, causing the *EMathError* exception to be destroyed. To the caller, it appears as if the *Tan* function has raised an *ETrigError* exception.

Try...finally statements

Sometimes you want to ensure that specific parts of an operation are completed, whether or not the operation is interrupted by an exception. For example, when a routine acquires control of a resource, it is often important that the resource be released, regardless of whether the routine terminates normally. In these situations, you can use a **try...finally** statement.

The following example shows how code that opens and processes a file can ensure that the file is ultimately closed, even if an error occurs during execution.

```
Reset(F);
try
 ⋮ // process file F
finally
  CloseFile(F);
end;
```

The syntax of a **try...finally** statement is

`try` *statementList*$_1$ `finally` *statementList*$_2$ `end`

where each *statementList* is a sequence of statements delimited by semicolons. The **try...finally** statement executes the statements in *statementList*$_1$ (the **try** clause). If *statementList*$_1$ finishes without raising exceptions, *statementList*$_2$ (the **finally** clause) is executed. If an exception is raised during execution of *statementList*$_1$, control is transferred to *statementList*$_2$; once *statementList*$_2$ finishes executing, the exception is re-raised. If a call to the *Exit*, *Break*, or *Continue* procedure causes control to leave *statementList*$_1$, *statementList*$_2$ is automatically executed. Thus the **finally** clause is always executed, regardless of how the **try** clause terminates.

If an exception is raised but not handled in the **finally** clause, that exception is propagated out of the **try...finally** statement, and any exception already raised in the **try** clause is lost. The **finally** clause should therefore handle all locally raised exceptions, so as not to disturb propagation of other exceptions.

Standard exception classes and routines

The *SysUtils* unit declares several standard routines for handling exceptions, including *ExceptObject*, *ExceptAddr*, and *ShowException*. *SysUtils* and other units also include dozens of exception classes, all of which (aside from *OutlineError*) derive from *Exception*.

The *Exception* class has properties called *Message* and *HelpContext* that can be used to pass an error description and a context ID for context-sensitive online documentation. It also defines various constructor methods that allow you to specify the description and context ID in different ways. See the online Help for details.

Chapter

8

Standard routines and I/O

This chapter discusses text and file I/O and summarizes standard library routines. Many of the procedures and functions listed here are defined in the *System* unit, which is implicitly compiled with every application. Others are built into the compiler but are treated as if they were in the *System* unit.

Some standard routines are in units such as *SysUtils*, which must be listed in a **uses** clause to make them available in programs. You cannot, however, list *System* in a **uses** clause, nor should you modify the *System* unit or try to rebuild it explicitly.

For more information about the routines listed here, see the online Help.

File input and output

The table below lists input and output routines.

Table 8.1 Input and output procedures and functions

Procedure or function	Description
Append	Opens an existing text file for appending.
AssignFile	Assigns the name of an external file to a file variable.
BlockRead	Reads one or more records from an untyped file.
BlockWrite	Writes one or more records into an untyped file.
ChDir	Changes the current directory.
CloseFile	Closes an open file.
Eof	Returns the end-of-file status of a file.
Eoln	Returns the end-of-line status of a text file.
Erase	Erases an external file.
FilePos	Returns the current file position of a typed or untyped file.
FileSize	Returns the current size of a file; not used for text files.

Table 8.1 Input and output procedures and functions (continued)

Procedure or function	Description
Flush	Flushes the buffer of an output text file.
GetDir	Returns the current directory of a specified drive.
IOResult	Returns an integer value that is the status of the last I/O function performed.
MkDir	Creates a subdirectory.
Read	Reads one or more values from a file into one or more variables.
Readln	Does what *Read* does and then skips to beginning of next line in the text file.
Rename	Renames an external file.
Reset	Opens an existing file.
Rewrite	Creates and opens a new file.
RmDir	Removes an empty subdirectory.
Seek	Moves the current position of a typed or untyped file to a specified component. Not used with text files.
SeekEof	Returns the end-of-file status of a text file.
SeekEoln	Returns the end-of-line status of a text file.
SetTextBuf	Assigns an I/O buffer to a text file.
Truncate	Truncates a typed or untyped file at the current file position.
Write	Writes one or more values to a file.
Writeln	Does the same as *Write*, and then writes an end-of-line marker to the text file.

A file variable is any variable whose type is a file type. There are three classes of file: *typed*, *text*, and *untyped*. The syntax for declaring file types is given in “File types” on page 5-25.

Before a file variable can be used, it must be associated with an external file through a call to the *AssignFile* procedure. An external file is typically a named disk file, but it can also be a device, such as the keyboard or the display. The external file stores the information written to the file or supplies the information read from the file.

Once the association with an external file is established, the file variable must be “opened” to prepare it for input or output. An existing file can be opened via the *Reset* procedure, and a new file can be created and opened via the *Rewrite* procedure. Text files opened with *Reset* are read-only and text files opened with *Rewrite* and *Append* are write-only. Typed files and untyped files always allow both reading and writing regardless of whether they were opened with *Reset* or *Rewrite*.

Every file is a linear sequence of components, each of which has the component type (or record type) of the file. The components are numbered starting with zero.

Files are normally accessed sequentially. That is, when a component is read using the standard procedure *Read* or written using the standard procedure *Write*, the current file position moves to the next numerically ordered file component. Typed files and untyped files can also be accessed randomly through the standard procedure *Seek*, which moves the current file position to a specified component. The standard functions *FilePos* and *FileSize* can be used to determine the current file position and the current file size.

When a program completes processing a file, the file must be closed using the standard procedure *CloseFile*. After a file is closed, its associated external file is updated. The file variable can then be associated with another external file.

By default, all calls to standard I/O procedures and functions are automatically checked for errors, and if an error occurs an exception is raised (or the program is terminated if exception handling is not enabled). This automatic checking can be turned on and off using the **{$I+}** and **{$I–}** compiler directives. When I/O checking is off—that is, when a procedure or function call is compiled in the **{$I–}** state—an I/O error doesn't cause an exception to be raised; to check the result of an I/O operation, you must call the standard function *IOResult* instead.

You must call the *IOResult* function to clear an error, even if you aren't interested in the error. If you don't clear an error and **{$I+}** is the current state, the next I/O function call will fail with the lingering *IOResult* error.

Text files

This section summarizes I/O using file variables of the standard type *Text*.

When a text file is opened, the external file is interpreted in a special way: It is considered to represent a sequence of characters formatted into lines, where each line is terminated by an end-of-line marker (a carriage-return character, possibly followed by a linefeed character). The type *Text* is distinct from the type `file of Char`.

For text files, there are special forms of *Read* and *Write* that let you read and write values that are not of type *Char*. Such values are automatically translated to and from their character representation. For example, `Read(F, I)`, where *I* is a type *Integer* variable, reads a sequence of digits, interprets that sequence as a decimal integer, and stores it in *I*.

There are two standard text file variables, *Input* and *Output*. The standard file variable *Input* is a read-only file associated with the operating system's standard input (typically, the keyboard). The standard file variable *Output* is a write-only file associated with the operating system's standard output (typically, the display). Before an application begins executing, *Input* and *Output* are automatically opened, as if the following statements were executed:

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

Note Text-oriented I/O is available only in console applications—that is, applications compiled with the "Generate console application" option checked on the Linker page of the Project Options dialog box or with the **-cc** command-line compiler option. In a GUI (non-console) application, any attempt to read or write using *Input* or *Output* will produce an I/O error.

Some of the standard I/O routines that work on text files don't need to have a file variable explicitly given as a parameter. If the file parameter is omitted, *Input* or *Output* is assumed by default, depending on whether the procedure or function is input- or output-oriented. For example, `Read(X)` corresponds to `Read(Input, X)` and `Write(X)` corresponds to `Write(Output, X)`.

If you do specify a file when calling one of the input or output routines that work on text files, the file must be associated with an external file using *AssignFile*, and opened using *Reset*, *Rewrite*, or *Append*. An exception is raised if you pass a file that was opened with *Reset* to an output-oriented procedure or function. An exception is also raised if you pass a file that was opened with *Rewrite* or *Append* to an input-oriented procedure or function.

Untyped files

Untyped files are low-level I/O channels used primarily for direct access to disk files regardless of type and structuring. An untyped file is declared with the word **file** and nothing more. For example,

```
var DataFile: file;
```

For untyped files, the *Reset* and *Rewrite* procedures allow an extra parameter to specify the record size used in data transfers. For historical reasons, the default record size is 128 bytes. A record size of 1 is the only value that correctly reflects the exact size of any file. (No partial records are possible when the record size is 1.)

Except for *Read* and *Write*, all typed-file standard procedures and functions are also allowed on untyped files. Instead of *Read* and *Write*, two procedures called *BlockRead* and *BlockWrite* are used for high-speed data transfers.

Text file device drivers

You can define your own text file device drivers for your programs. A text file device driver is a set of four functions that completely implement an interface between Object Pascal's file system and some device.

The four functions that define each device driver are *Open*, *InOut*, *Flush*, and *Close*. The function header of each function is

```
function DeviceFunc(var F: TTextRec): Integer;
```

where *DeviceFunc* is the name of the function (that is, *Open*, *InOut*, *Flush*, or *Close*). For information about the *TTextRec* type, see the online Help. The return value of a device-interface function becomes the value returned by *IOResult*. If the return value is zero, the operation was successful.

To associate the device-interface functions with a specific file, you must write a customized *Assign* procedure. The *Assign* procedure must assign the addresses of the four device-interface functions to the four function pointers in the text file variable. In addition, it should store the *fmClosed* "magic" constant in the *Mode* field, store the size of the text file buffer in *BufSize*, store a pointer to the text file buffer in *BufPtr*, and clear the *Name* string.

Assuming, for example, that the four device-interface functions are called *DevOpen*, *DevInOut*, *DevFlush*, and *DevClose*, the *Assign* procedure might look like this:

```
procedure AssignDev(var F: Text);
begin
  with TTextRec(F) do
  begin
    Mode := fmClosed;
    BufSize := SizeOf(Buffer);
    BufPtr := @Buffer;
    OpenFunc := @DevOpen;
    InOutFunc := @DevInOut;
    FlushFunc := @DevFlush;
    CloseFunc := @DevClose;
    Name[0] := #0;
  end;
end;
```

The device-interface functions can use the *UserData* field in the file record to store private information. This field isn't modified by the product file system at any time.

Device functions

The functions that make up a text file device driver are described below.

The Open function

The *Open* function is called by the *Reset*, *Rewrite*, and *Append* standard procedures to open a text file associated with a device. On entry, the *Mode* field contains *fmInput*, *fmOutput*, or *fmInOut* to indicate whether the *Open* function was called from *Reset*, *Rewrite*, or *Append*.

The *Open* function prepares the file for input or output, according to the *Mode* value. If *Mode* specified *fmInOut* (indicating that *Open* was called from *Append*), it must be changed to *fmOutput* before *Open* returns.

Open is always called before any of the other device-interface functions. For that reason, *AssignDev* only initializes the *OpenFunc* field, leaving initialization of the remaining vectors up to *Open*. Based on *Mode*, *Open* can then install pointers to either input- or output-oriented functions. This saves the *InOut*, *Flush* functions and the *CloseFile* procedure from determining the current mode.

The InOut function

The *InOut* function is called by the *Read*, *Readln*, *Write*, *Writeln*, *Eof*, *Eoln*, *SeekEof*, *SeekEoln*, and *CloseFile* standard routines whenever input or output from the device is required.

When *Mode* is *fmInput*, the *InOut* function reads up to *BufSize* characters into `BufPtr^`, and returns the number of characters read in *BufEnd*. In addition, it stores zero in *BufPos*. If the *InOut* function returns zero in *BufEnd* as a result of an input request, *Eof* becomes *True* for the file.

When *Mode* is *fmOutput*, the *InOut* function writes *BufPos* characters from `BufPtr^`, and returns zero in *BufPos*.

The Flush function

The *Flush* function is called at the end of each *Read*, *Readln*, *Write*, and *Writeln*. It can optionally flush the text file buffer.

If *Mode* is *fmInput*, the *Flush* function can store zero in *BufPos* and *BufEnd* to flush the remaining (unread) characters in the buffer. This feature is seldom used.

If *Mode* is *fmOutput*, the *Flush* function can write the contents of the buffer exactly like the *InOut* function, which ensures that text written to the device appears on the device immediately. If *Flush* does nothing, the text doesn't appear on the device until the buffer becomes full or the file is closed.

The Close function

The *Close* function is called by the *CloseFile* standard procedure to close a text file associated with a device. (The *Reset*, *Rewrite*, and *Append* procedures also call *Close* if the file they are opening is already open.) If *Mode* is *fmOutput*, then before calling *Close*, the file system calls the *InOut* function to ensure that all characters have been written to the device.

Handling null-terminated strings

Object Pascal's extended syntax allows the *Read*, *Readln*, *Str*, and *Val* standard procedures to be applied to zero-based character arrays, and allows the *Write*, *Writeln*, *Val*, *AssignFile*, and *Rename* standard procedures to be applied to both zero-based character arrays and character pointers. In addition, the following functions are provided for handling null-terminated strings. For more information about null-terminated strings, see "Working with null-terminated strings" on page 5-13.

Table 8.2 Null-terminated string functions

Function	Description
StrAlloc	Allocates a character buffer of a given size on the heap.
StrBufSize	Returns the size of a character buffer allocated using *StrAlloc* or *StrNew*.
StrCat	Concatenates two strings.
StrComp	Compares two strings.
StrCopy	Copies a string.
StrDispose	Disposes a character buffer allocated using *StrAlloc* or *StrNew*.
StrECopy	Copies a string and returns a pointer to the end of the string.
StrEnd	Returns a pointer to the end of a string.
StrFmt	Formats one or more values into a string.
StrIComp	Compares two strings without case sensitivity.
StrLCat	Concatenates two strings with a given maximum length of the resulting string.
StrLComp	Compares two strings for a given maximum length.
StrLCopy	Copies a string up to a given maximum length.
StrLen	Returns the length of a string.
StrLFmt	Formats one or more values into a string with a given maximum length.

Table 8.2 Null-terminated string functions (continued)

Function	Description
StrLIComp	Compares two strings for a given maximum length without case sensitivity.
StrLower	Converts a string to lowercase.
StrMove	Moves a block of characters from one string to another.
StrNew	Allocates a string on the heap.
StrPCopy	Copies a Pascal string to a null-terminated string.
StrPLCopy	Copies a Pascal string to a null-terminated string with a given maximum length.
StrPos	Returns a pointer to the first occurrence of a given substring within a string.
StrRScan	Returns a pointer to the last occurrence of a given character within a string.
StrScan	Returns a pointer to the first occurrence of a given character within a string.
StrUpper	Converts a string to uppercase.

Standard string-handling functions have multibyte-enabled counterparts that also implement locale-specific ordering for characters. Names of multibyte functions start with *Ansi-*. For example, the multibyte version of *StrPos* is *AnsiStrPos*. Multibyte character support is operating-system dependent and based on the current locale.

Wide-character strings

The *System* unit provides three functions, *WideCharToString*, *WideCharLenToString*, and *StringToWideChar*, that can be used to convert null-terminated wide character strings to single- or double-byte long strings.

For more information about wide-character strings, see “About extended character sets” on page 5-13.

Other standard routines

The table below lists frequently used procedures and functions found in Borland product libraries. This is not an exhaustive inventory of standard routines. For more information about these and other routines, see the online Help.

Table 8.3 Other standard routines

Procedure or function	Description
Abort	Ends the process without reporting an error.
Addr	Returns a pointer to a specified object.
AllocMem	Allocates a memory block and initializes each byte to zero.
ArcTan	Calculates the arctangent of the given number.
Assert	Tests whether a boolean expression is *True*.
Assigned	Tests for a **nil** (unassigned) pointer or procedural variable.
Beep	Generates a standard beep using the computer speaker.

Table 8.3 Other standard routines (continued)

Procedure or function	Description
Break	Causes control to exit a **for**, **while**, or **repeat** statement.
ByteToCharIndex	Returns the position of the character containing a specified byte in a string.
Chr	Returns the character for a specified value.
Close	Terminates the association between a file variable and an external file.
CompareMem	Performs a binary comparison of two memory images.
CompareStr	Compares strings case sensitively.
CompareText	Compares strings by ordinal value and is not case sensitive.
Continue	Returns control to the next iteration of **for**, **while**, or **repeat** statements.
Copy	Returns a substring of a string or a segment of a dynamic array.
Cos	Calculates the cosine of an angle.
CurrToStr	Converts a currency variable to a string.
Date	Returns the current date.
DateTimeToStr	Converts a variable of type *TDateTime* to a string.
DateToStr	Converts a variable of type *TDateTime* to a string.
Dec	Decrements an ordinal variable.
Dispose	Releases memory allocated for a dynamic variable.
ExceptAddr	Returns the address at which the current exception was raised.
Exit	Exits from the current procedure.
Exp	Calculates the exponential of X.
FillChar	Fills contiguous bytes with a specified value.
Finalize	Uninitializes a dynamically allocated variable.
FloatToStr	Converts a floating point value to a string.
FloatToStrF	Converts a floating point value to a string, using specified format.
FmtLoadStr	Returns formatted output using a resourced format string.
FmtStr	Assembles a formatted string from a series of arrays.
Format	Assembles a string from a format string and a series of arrays.
FormatDateTime	Formats a date-and-time value.
FormatFloat	Formats a floating point value.
FreeMem	Disposes of a dynamic variable.
GetMem	Creates a dynamic variable and a pointer to the address of the block.
GetParentForm	Returns the form or property page that contains a specified control.
Halt	Initiates abnormal termination of a program.
Hi	Returns the high-order byte of an expression as an unsigned value.
High	Returns the highest value in the range of a type, array, or string.
Inc	Increments an ordinal variable.
Initialize	Initializes a dynamically allocated variable.
Insert	Inserts a substring at a specified point in a string.
Int	Returns the integer part of a real number.
IntToStr	Converts an integer to a string.

Table 8.3 Other standard routines (continued)

Procedure or function	Description
Length	Returns the length of a string or array.
Lo	Returns the low-order byte of an expression as an unsigned value.
Low	Returns the lowest value in the range of a type, array, or string.
LowerCase	Converts an ASCII string to lowercase.
MaxIntValue	Returns the largest signed value in an integer array.
MaxValue	Returns the largest signed value in an array.
MinIntValue	Returns the smallest signed value in an integer array.
MinValue	Returns smallest signed value in an array.
New	Creates a new dynamic variable and references it with a specified pointer.
Now	Returns the current date and time.
Ord	Returns the ordinal value of an ordinal-type expression.
Pos	Returns the index of the first character of a specified substring in a string.
Pred	Returns the predecessor of an ordinal value.
Ptr	Converts a specified address to a pointer.
Random	Generates random numbers within a specified range.
ReallocMem	Reallocates a dynamic variable.
Round	Returns the value of a real rounded to the nearest whole number.
SetLength	Sets the dynamic length of a string variable or array.
SetString	Sets the contents and length of the given string.
ShowException	Displays an exception message with its address.
ShowMessage	Displays a message box with an unformatted string and an OK button.
ShowMessageFmt	Displays a message box with a formatted string and an OK button.
Sin	Returns the sine of an angle in radians.
SizeOf	Returns the number of bytes occupied by a variable or type.
Sqr	Returns the square of a number.
Sqrt	Returns the square root of a number.
Str	Formats a string and returns it to a variable.
StrToCurr	Converts a string to a currency value.
StrToDate	Converts a string to a date format (*TDateTime*).
StrToDateTime	Converts a string to a *TDateTime*.
StrToFloat	Converts a string to a floating-point value.
StrToInt	Converts a string to an integer.
StrToTime	Converts a string to a time format (*TDateTime*).
StrUpper	Returns a string in upper case.
Succ	Returns the successor of an ordinal value.
Sum	Returns the sum of the elements from an array.
Time	Returns the current time.
TimeToStr	Converts a variable of type *TDateTime* to a string.
Trunc	Truncates a real number to an integer.

Table 8.3 Other standard routines (continued)

Procedure or function	Description
UniqueString	Ensures that a string has only one reference. (The string may be copied to produce a single reference.)
UpCase	Converts a character to uppercase.
UpperCase	Returns a string in uppercase.
VarArrayCreate	Creates a variant array.
VarArrayDimCount	Returns number of dimensions of a variant array.
VarArrayHighBound	Returns high bound for a dimension in a variant array.
VarArrayLock	Locks a variant array and returns a pointer to the data.
VarArrayLowBound	Returns the low bound of a dimension in a variant array.
VarArrayOf	Creates and fills a one-dimensional variant array.
VarArrayRedim	Resizes a variant array.
VarArrayRef	Returns a reference to the passed variant array.
VarArrayUnlock	Unlocks a variant array.
VarAsType	Converts a variant to specified type.
VarCast	Converts a variant to a specified type, storing the result in a variable.
VarClear	Clears a variant.
VarCopy	Copies a variant.
VarToStr	Converts variant to string.
VarType	Returns type code of specified variant.

For information on format strings, see “Format strings” in the online Help.

Part

II

Special topics

The chapters in Part II cover specialized language features and advanced topics. These chapters include:

- Chapter 9, “Libraries and packages”
- Chapter 10, “Object interfaces”
- Chapter 11, “Memory management”
- Chapter 12, “Program control”
- Chapter 13, “Inline assembly code”

Chapter

9

Libraries and packages

A dynamically loadable library is a dynamic-link library (DLL) on Windows or a shared object library file on Linux. It is a collection of routines that can be called by applications and by other DLLs or shared objects. Like units, dynamically loadable libraries contain sharable code or resources. But this type of library is a separately compiled executable that is linked at runtime to the programs that use it.

To distinguish them from stand-alone executables, on Windows files containing compiled DLLs are named with the .DLL extension. On Linux, files containing shared object files are named with a .so extension. Object Pascal programs can call DLLs or shared objects written in other languages, and applications written in other languages can call DLLs or shared objects written in Object Pascal.

Calling dynamically loadable libraries

You can call operating system routines directly, but they are not linked to your application until runtime. This means that the library need not be present when you compile your program. It also means that there is no compile-time validation of attempts to import a routine.

Before you can call routines defined in a shared object, you must *import* them. This can be done in two ways: by declaring an **external** procedure or function, or by direct calls to the operating system. Whichever method you use, the routines are not linked to your application until runtime.

Object Pascal does not support importing of variables from shared libraries.

Static loading

The simplest way to import a procedure or function is to declare it using the **external** directive. For example,

On Windows: **procedure** DoSomething; **external** 'MYLIB.DLL';

On Linux: **procedure** DoSomething; **external** 'mylib.so';

If you include this declaration in a program, MYLIB.DLL (Windows) or mylib.so (Linux) is loaded once, when the program starts. Throughout execution of the program, the identifier *DoSomething* always refers to the same entry point in the same shared library.

Declarations of imported routines can be placed directly in the program or unit where they are called. To simplify maintenance, however, you can collect **external** declarations into a separate “import unit” that also contains any constants and types required for interfacing with the library. Other modules that use the import unit can call any routines declared in it.

For more information about **external** declarations, see “External declarations” on page 6-6.

Dynamic loading

You can access routines in a library through direct calls to OS library functions, including *LoadLibrary*, *FreeLibrary*, and *GetProcAddress.* In Windows, these functions are declared in Windows.pas; on Linux, they are implemented for compatibility in SysUtils.pas; the actual Linux OS routines are *dlopen*, *dlclose*, and *dlsym* (all declared in Kylix’s *Libc* unit; see the man pages for more information). In this case, use procedural-type variables to reference the imported routines.

For example, on Windows or Linux:

```
uses Windows, ...; {On Linux, replace Windows with SysUtils }

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;

  TGetTime = procedure(var Time: TTimeRec);
  THandle = Integer;

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  f
begin
  Handle := LoadLibrary('libraryname');
  if Handle <> 0 then
  begin
    @GetTime := GetProcAddress(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
    end;
    FreeLibrary(Handle);
  end;
end;
```

When you import routines this way, the library is not loaded until the code containing the call to *LoadLibrary* executes. The library is later unloaded by the call to *FreeLibrary*. This allows you to conserve memory and to run your program even when some of the libraries it uses are not present.

This same example can also be written on Linux as follows:

```
uses Libc, ...;

type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;

  TGetTime = procedure(var Time: TTimeRec);
  THandle = Pointer;

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  ƒ
begin
  Handle := dlopen('datetime.so', RTLD_LAZY);
  if Handle <> 0 then
  begin
    @GetTime := dlsym(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
    end;
    dlclose(Handle);
  end;
end;
```

In this case, when importing routines, the shared object is not loaded until the code containing the call to *dlopen* executes. The shared object is later unloaded by the call to *dlclose*. This also allows you to conserve memory and to run your program even when some of the shared objects it uses are not present.

Writing dynamically loadable libraries

The main source for a dynamically loadable library is identical to that of a program, except that it begins with the reserved word **library** (instead of **program**).

Only routines that a library explicitly exports are available for importing by other libraries or programs. The following example shows a library with two exported functions, *Min* and *Max*.

```
library MinMax;

function Min(X, Y: Integer): Integer; stdcall;
begin
  if X < Y then Min := X else Min := Y;
end;

function Max(X, Y: Integer): Integer; stdcall;
begin
  if X > Y then Max := X else Max := Y;
end;

exports
  Min,
  Max;

begin
end.
```

If you want your library to be available to applications written in other languages, it's safest to specify **stdcall** in the declarations of exported functions. Other languages may not support Object Pascal's default **register** calling convention.

Libraries can be built from multiple units. In this case, the library source file is frequently reduced to a **uses** clause, an **exports** clause, and the initialization code. For example,

```
library Editors;

uses EdInit, EdInOut, EdFormat, EdPrint;

exports
  InitEditors,
  DoneEditors name Done,
  InsertText name Insert,
  DeleteSelection name Delete,
  FormatSelection,
  PrintSelection name Print,
  ƒ
  SetErrorHandler;

begin
  InitLibrary;
end.
```

You can put **exports** clauses in the **interface** or **implementation** section of a unit. Any library that includes such a unit in its **uses** clause automatically exports the routines listed the unit's **exports** clauses—without the need for an **exports** clause of its own.

The directive **local**, which marks routines as unavailable for export, is platform-specific and has no effect in Windows programming.

On Linux, the **local** directive provides a slight performance optimization for routines that are compiled into a library but are not exported. This directive can be specified for stand-alone procedures and functions, but not for methods. A routine declared with **local**—for example,

```
function Contraband(I: Integer): Integer; local;
```

—does not refresh the EBX register and hence

- cannot be exported from a library.
- cannot be declared in the **interface** section of a unit.
- cannot have its address taken or be assigned to a procedural-type variable.
- if it is a pure assembler routine, cannot be called from another unit unless the *caller* sets up EBX.

The exports clause

A routine is exported when it is listed in an **exports** clause, which has the form

exports $entry_1$, ..., $entry_n$;

where each *entry* consists of the name of a procedure, function, or variable (which must be declared prior to the **exports** clause), followed by a parameter list (only if exporting a routine that is overloaded), and an optional **name** specifier. You can qualify the procedure or function name with the name of a unit.

(Entries can also include the directive **resident**, which is maintained for backward compatibility and is ignored by the compiler.)

On Windows only, an **index** specifier consists of the directive **index** followed by a numeric constant between 1 and 2,147,483,647. (For more efficient programs, use low index values.) If an entry has no **index** specifier, the routine is automatically assigned a number in the export table.

Note Use of **index** specifiers, which are supported for backward compatibility only, is discouraged and may cause problems for other development tools.

A **name** specifier consists of the directive **name** followed by a string constant. If an entry has no **name** specifier, the routine is exported under its original declared name, with the same spelling and case. Use a **name** clause when you want to export a routine under a different name. For example,

```
exports
  DoSomethingABC name 'DoSomething';
```

When you export an overloaded function or procedure from a dynamically loadable library, you must specify its parameter list in the **exports** clause. For example,

```
exports
  Divide(X, Y: Integer) name 'Divide_Ints',
  Divide(X, Y: Real) name 'Divide_Reals';
```

On Windows, do not include **index** specifiers in entries for overloaded routines.

An **exports** clause can appear anywhere and any number of times in the declaration part of a program or library, or in the **interface** or **implementation** section of a unit. Programs seldom contain an **exports** clause.

Library initialization code

The statements in a library's block constitute the library's *initialization code*. These statements are executed once every time the library is loaded. They typically perform tasks like registering window classes and initializing variables. Library initialization code can also install an entry point procedure using the *DllProc* variable. The *DllProc* variable is similar to an exit procedure, which is described in "Exit procedures" on page 12-4; the entry point procedure executes when the library is loaded or unloaded.

Library initialization code can signal an error by setting the *ExitCode* variable to a nonzero value. *ExitCode* is declared in the *System* unit and defaults to zero, indicating successful initialization. If a library's initialization code sets *ExitCode* to another value, the library is unloaded and the calling application is notified of the failure. Similarly, if an unhandled exception occurs during execution of the initialization code, the calling application is notified of a failure to load the library.

Here is an example of a library with initialization code and an entry point procedure.

```
library Test;

var
  SaveDllProc: Pointer;

procedure LibExit(Reason: Integer);
begin
  if Reason = DLL_PROCESS_DETACH then
  begin
  ƒ  // library exit code
  end;
  SaveDllProc(Reason);  // call saved entry point procedure
end;

begin
  ƒ  // library initialization code
  SaveDllProc := DllProc;  // save exit procedure chain
  DllProc := @LibExit;  // install LibExit exit procedure
end.
```

DllProc is called when the library is first loaded into memory, when a thread starts or stops, or when the library is unloaded. The initialization parts of all units used by a library are executed before the library's initialization code, and the finalization parts of those units are executed after the library's entry point procedure.

Global variables in a library

Global variables declared in a shared library cannot be imported by an Object Pascal application.

A library can be used by several applications at once, but each application has a copy of the library in its own process space with its own set of global variables. For multiple libraries—or multiple instances of a library—to share memory, they must use memory-mapped files. Refer to the your system documentation for further information.

Libraries and system variables

Several variables declared in the *System* unit are of special interest to those programming libraries. Use *IsLibrary* to determine whether code is executing in an application or in a library; *IsLibrary* is always *False* in an application and *True* in a library. During a library's lifetime, *HInstance* contains its instance handle. *CmdLine* is always **nil** in a library.

The *DLLProc* variable allows a library to monitor calls that the operating system makes to the library entry point. This feature is normally used only by libraries that support multithreading. *DLLProc* is available on both Windows and Linux but its use differs on each. On Windows, *DLLProc* is used in multithreading applications; on Linux, it is used to determine when your library is being unloaded. You should use finalization sections, rather than exit procedures, for all exit behavior. (See "The finalization section" on page 3-5.)

To monitor operating-system calls, create a callback procedure that takes a single integer parameter—for example,

```
procedure DLLHandler(Reason: Integer);
```

—and assign the address of the procedure to the *DLLProc* variable. When the procedure is called, it passes to it one of the following values.

DLL_PROCESS_DETACH	Indicates that the library is detaching from the address space of the calling process as a result of a clean exit or a call to *FreeLibrary* (*dlclose* on Linux).
DLL_PROCESS_ATTACH	Indicates that the library is attaching to the address space of the calling process as the result of a call to *LoadLibrary* (*dlopen* on Linux).
DLL_THREAD_ATTACH	Indicates that the current process is creating a new thread (Windows only).
DLL_THREAD_DETACH	Indicates that a thread is exiting cleanly (Windows only).

In the body of the procedure, you can specify actions to take depending on which parameter is passed to the procedure.

Exceptions and runtime errors in libraries

When an exception is raised but not handled in a dynamically loadable library, it propagates out of the library to the caller. If the calling application or library is itself written in Object Pascal, the exception can be handled through a normal **try...except** statement.

Note Under Linux this is only possible if the library and application have both been built with the same set of (base) runtime packages (which contains the EH code) or if both link to ShareExcept.

On Windows, If the calling application or library is written in another language, the exception can be handled as an operating-system exception with the exception code $0EEDFADE. The first entry in the *ExceptionInformation* array of the operating-system exception record contains the exception address, and the second entry contains a reference to the Object Pascal exception object.

Generally, you should not let exceptions escape from your library. On Windows, Delphi exceptions map to the OS exception model; Linux does not have an exception model.

If a library does not use the *SysUtils* unit, exception support is disabled. In this case, when a runtime error occurs in the library, the calling application terminates. Because the library has no way of knowing whether it was called from an Object Pascal program, it cannot invoke the application's exit procedures; the application is simply aborted and removed from memory.

Shared-memory manager (Windows only)

On Windows, if a DLL exports routines that pass long strings or dynamic arrays as parameters or function results (whether directly or nested in records or objects), then the DLL and its client applications (or DLLs) must all use the *ShareMem* unit. The same is true if one application or DLL allocates memory with *New* or *GetMem* which is deallocated by a call to *Dispose* or *FreeMem* in another module. *ShareMem* should always be the first unit listed in any program or library **uses** clause where it occurs.

ShareMem is the interface unit for the BORLANDMM.DLL memory manager, which allows modules to share dynamically allocated memory. BORLANDMM.DLL must be deployed with applications and DLLs that use *ShareMem*. When an application or DLL uses *ShareMem*, its memory manager is replaced by the memory manager in BORLANDMM.DLL.

Linux uses glibc's *malloc* to manage shared memory.

Packages

A package is a specially compiled library used by applications, the IDE, or both. Packages allow you to rearrange when code resides without affecting the source code. This is sometimes referred to as *application partitioning*.

Runtime packages provide functionality when a user runs an application. *Design-time packages* are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by referencing runtime packages in their **requires** clauses.

To distinguish them from other libraries, packages are stored in files:

- On Windows, package files end with the .bpl (Borland package library) extension.
- On Linux, packages generally begin with the prefix bpl and have a .so extension.

Ordinarily, packages are loaded statically when an applications starts. But you can use the *LoadPackage* and *UnloadPackage* routines (in the *SysUtils* unit) to load packages dynamically.

Note When an application utilizes packages, the name of each packaged unit still must appear in the **uses** clause of any source file that references it. For more information about packages, see the online Help.

Package declarations and source files

Each package is declared in a separate source file, which should be saved with the .dpk extension to avoid confusion with other files containing Object Pascal code. A package source file does not contain type, data, procedure, or function declarations. Instead, it contains

- A *name* for the package.
- A list of other packages *required* by the new package. These are packages to which the new package is linked.
- A list of unit files *contained* by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units, which provide the functionality of the compiled package.

A package declaration has the form

```
package packageName;
  requiresClause;
  containsClause;
end.
```

where *packageName* is any valid identifier. The *requiresClause* and *containsClause* are both optional. For example, the following code declares the *DATAX* package.

```
package DATAX;
requires
  baseclx,
  visualclx;
  contains Db, DBLocal, DBXpress, ... ;
end.
```

The **requires** clause lists other, external packages used by the package being declared. It consists of the directive **requires**, followed by a comma-delimited list of package names, followed by a semicolon. If a package does not reference other packages, it does not need a **requires** clause.

The **contains** clause identifies the unit files to be compiled and bound into the package. It consists of the directive **contains**, followed by a comma-delimited list of unit names, followed by a semicolon. Any unit name may be followed by the reserved word **in** and the name of a source file, with or without a directory path, in single quotation marks; directory paths can be absolute or relative. For example,

```
contains MyUnit in 'C:\MyProject\MyUnit.pas'; // Windows
```

```
contains MyUnit in '\home\developer\MyProject\MyUnit.pas'; // Linux
```

Note Thread-local variables (declared with **threadvar**) in a packaged unit cannot be accessed from clients that use the package.

Naming packages

A compiled package involves several generated files. For example, the source file for the package called *DATAX* is DATAX.dpk, from which the compiler generates an executable and a binary image called

- On Windows: DATAX.bpl and DATAX.dcp
- On Linux: bplDATAX.so and DATAX.dcp.

DATAX is used to refer to the package in the **requires** clauses of other packages, or when using the package in an application. Package names must be unique within a project.

The requires clause

The **requires** clause lists other, external packages that are used by the current package. It functions like the **uses** clause in a unit file. An external package listed in the **requires** clause is automatically linked at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in a package make references to other packaged units, the other packages should be included in the first package's **requires** clause. If the other packages are omitted from the **requires** clause, the compiler loads the referenced units from their .dcu (Windows) or .dpu (Linux) files.

Avoiding circular package references

Packages cannot contain circular references in their **requires** clauses. This means that

- A package cannot reference itself in its own **requires** clause.
- A chain of references must terminate without rereferencing any package in the chain. If package *A* requires package *B*, then package *B* cannot require package *A*; if package *A* requires package *B* and package *B* requires package *C*, then package *C* cannot require package *A*.

Duplicate package references

The compiler ignores duplicate references in a package's **requires** clause. For programming clarity and readability, however, duplicate references should be removed.

The contains clause

The **contains** clause identifies the unit files to be bound into the package. Do not include file-name extensions in the **contains** clause.

Avoiding redundant source code uses

A package cannot be listed in the **contains** clause of another package or the **uses** clause of a unit.

All units included directly in a package's **contains** clause, or indirectly in the **uses** clauses of those units, are bound into the package at compile time. The units contained (directly or indirectly) in a package cannot be contained in any other packages referenced in **requires** clause of that package.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application.

Compiling packages

Packages are ordinarily compiled from the IDE using .dpk files generated by the Package editor. You can also compile .dpk files directly from the command line. When you build a project that contains a package, the package is implicitly recompiled, if necessary.

Generated files

The following table lists the files produced by the successful compilation of a package.

Table 9.1 Compiled package files

File extension	Contents
dcp	A binary image containing a package header and the concatenation of all dcu (Windows) or dpu (Linux) files in the package. A single dcp file is created for each package. The base name for the dcp is the base name of the dpk source file.
dcu (Windows) dpu (Linux)	A binary image for a unit file contained in a package. One dcu or dpu file is created, when necessary, for each unit file.
.bpl on Windows bpl<package>.so on Linux	The runtime package. This file is a shared library with special Borland-specific features. The base name for the package is the base name of the dpk source file.

Several compiler directives and command-line switches are available to support package compilation.

Package-specific compiler directives

The following table lists package-specific compiler directives that can be inserted into source code. See the online Help for details.

Table 9.2 Package-specific compiler directives

Directive	Purpose
{$IMPLICITBUILD OFF}	Prevents a package from being implicitly recompiled later. Use in .dpk files when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.
{$G–} or {$IMPORTEDDATA OFF}	Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages.
{$WEAKPACKAGEUNIT ON}	Packages unit "weakly", as explained in the online Help.
{$DENYPACKAGEUNIT ON}	Prevents unit from being placed in a package.
{$DESIGNONLY ON}	Compiles the package for installation in the IDE. (Put in .dpk file.)
{$RUNONLY ON}	Compiles the package as runtime only. (Put in .dpk file.)

Including **{$DENYPACKAGEUNIT ON}** in source code prevents the unit file from being packaged. Including **{$G–}** or **{$IMPORTEDDATA OFF}** may prevent a package from being used in the same application with other packages.

Other compiler directives may be included, if appropriate, in package source code.

Package-specific command-line compiler switches

The following package-specific switches are available for the command-line compiler. See the online Help for details.

Table 9.3 Package-specific command-line compiler switches

Switch	Purpose
–$G–	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
–LE *path*	Specifies the directory where the compiled package file will be placed.
–LN *path*	Specifies the directory where the package dcp file will be placed.
–LU*packageName [;packageName2;...]*	Specifies additional runtime packages to use in an application. Used when compiling a project.
–Z	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

Using the **–$G–** switch may prevent a package from being used in the same application with other packages.

Other command-line options may be used, if appropriate, when compiling packages.

Chapter

10

Object interfaces

An *object interface*—or simply *interface*—defines methods that can be implemented by a class. Interfaces are declared like classes, but cannot be directly instantiated and do not have their own method definitions. Rather, it is the responsibility of any class that supports an interface to provide implementations for the interface's methods. A variable of an interface type can reference an object whose class implements that interface; however, only methods declared in the interface can be called using such a variable.

Interfaces offer some of the advantages of multiple inheritance without the semantic difficulties. They are also essential for using distributed object models. Custom objects built that support interfaces can interact with objects written in C++, Java, and other languages.

Interface types

Interfaces, like classes, can be declared only in the outermost scope of a program or unit, not in a procedure or function declaration. An interface type declaration has the form

```
type interfaceName = interface (ancestorInterface)
  ['{GUID}']
  memberList
end;
```

where `(`*ancestorInterface*`)` and `['{`*GUID*`}']` are optional. In most respects, interface declarations resemble class declarations, but the following restrictions apply.

- The *memberList* can include only methods and properties. Fields are not allowed in interfaces.
- Since an interface has no fields, property **read** and **write** specifiers must be methods.

- All members of an interface are public. Visibility specifiers and storage specifiers are not allowed. (But an array property can be declared as **default**.)
- Interfaces have no constructors or destructors. They cannot be instantiated, except through classes that implement their methods.
- Methods cannot be declared as **virtual**, **dynamic**, **abstract**, or **override**. Since interfaces do not implement their own methods, these designations have no meaning.

Here is an example of an interface declaration:

```
type
  IMalloc = interface(IInterface)
    ['{00000002-0000-0000-C000-000000000046}']
    function Alloc(Size: Integer): Pointer; stdcall;
    function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
    procedure Free(P: Pointer); stdcall;
    function GetSize(P: Pointer): Integer; stdcall;
    function DidAlloc(P: Pointer): Integer; stdcall;
    procedure HeapMinimize; stdcall;
  end;
```

In some interface declarations, the **interface** reserved word is replaced by **dispinterface**. This construction (along with the **dispid**, **read only**, and **write only** directives) is platform-specific and is not used in Linux programming.

IInterface and inheritance

An interface, like a class, inherits all of its ancestors' methods. But interfaces, unlike classes, do not *implement* methods. What an interface inherits is the *obligation* to implement methods—an obligation that devolves onto any class supporting the interface.

The declaration of an interface can specify an ancestor interface. If no ancestor is specified, the interface is a direct descendant of *IInterface*, which is defined in the *System* unit and is the ultimate ancestor of all other interfaces. *IInterface* declares three methods: *QueryInterface*, *_AddRef*, and *_Release*.

Note *IInterface* is equivalent to *IUnknown.* You should generally use *IInterface* for platform independent applications and reserve the use of *IUnknown* for specific programs that include Windows dependencies.

QueryInterface provides the means to move freely among the different interfaces that an object supports. *_AddRef* and *_Release* provide lifetime memory management for interface references. The easiest way to implement these methods is to derive the implementing class from the *System* unit's *TInterfacedObject*. It is also possible to dispense with any of these methods by implementing it as an empty function; COM objects (Windows only), however, must be managed through *_AddRef* and *_Release*.

Interface identification

An interface declaration can specify a globally unique identifier (GUID), represented by a string literal enclosed in brackets immediately preceding the member list. The GUID part of the declaration must have the form

```
['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
```

where each *x* is a hexadecimal digit (0 through 9 or A through F). On Windows, the Type Library editor automatically generates GUIDs for new interfaces; you can also generate GUIDs by pressing *Ctrl+Shift+G* in the Code editor (on Linux, you must use *Ctrl+Shift+G*).

A GUID is a 16-byte binary value that uniquely identifies an interface. If an interface has a GUID, you can use interface querying to get references to its implementations. (See “Interface querying” on page 10-10.)

The *TGUID* and *PGUID* types, declared in the *System* unit, are used to manipulate GUIDs.

```
type
  PGUID = ^TGUID;
  TGUID = packed record
    D1: Longword;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

When you declare a typed constant of type *TGUID*, you can use a string literal to specify its value. For example,

```
const IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

In procedure and function calls, either a GUID or an interface identifier can serve as a value or constant parameter of type *TGUID*. For example, given the declaration

```
function Supports(Unknown: IInterface; const IID: TGUID): Boolean;
```

Supports can be called in either of two ways:

```
if Supports(Allocator, IMalloc) then ...
if Supports(Allocator, IID_IMalloc) then ...
```

Calling conventions for interfaces

The default calling convention is **register**, but interfaces shared among modules (especially if they are written in different languages) should declare all methods with **stdcall**. Use **safecall** to implement CORBA interfaces. On Windows, you can use **safecall** to implement methods of dual interfaces (as described in “Dual interfaces (Windows only)” on page 10-13).

For more information about calling conventions, see “Calling conventions” on page 6-4.

Interface properties

Properties declared in an interface are accessible only through expressions of the interface type; they cannot be accessed through class-type variables. Moreover, interface properties are visible only within programs where the interface is compiled. For example, on Windows, COM objects do not have properties.

In an interface, property **read** and **write** specifiers must be methods, since fields are not available.

Forward declarations

An interface declaration that ends with the reserved word **interface** and a semicolon, without specifying an ancestor, GUID, or member list, is a *forward declaration*. A forward declaration must be resolved by a *defining declaration* of the same interface within the same type declaration section. In other words, between a forward declaration and its defining declaration, nothing can occur except other type declarations.

Forward declarations allow mutually dependent interfaces. For example,

```
type
  IControl = interface;
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000044}']
    function GetControl(Index: Integer): IControl;
    ⋮
  end;
  IControl = interface
    ['{00000115-0000-0000-C000-000000000049}']
    function GetWindow: IWindow;
    ⋮
  end;
```

Mutually derived interfaces are not allowed. For example, it is not legal to derive *IWindow* from *IControl* and also derive *IControl* from *IWindow*.

Implementing interfaces

Once an interface has been declared, it must be implemented in a class before it can be used. The interfaces implemented by a class are specified in the class's declaration, after the name of the class's ancestor. Such declarations have the form

```
type className = class (ancestorClass, interface1, ..., interfacen)
  memberList
 end;
```

For example,

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    ⋮
  end;
```

declares a class called *TMemoryManager* that implements the *IMalloc* and *IErrorInfo* interfaces. When a class implements an interface, it must implement (or inherit an implementation of) each method declared in the interface.

Here is the declaration of *TInterfacedObject* in the *System* unit.

```
type
TInterfacedObject = class(TObject, IInterface)
protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;
```

TInterfacedObject implements the *IInterface* interface. Hence *TInterfacedObject* declares and implements each of *IInterface*'s three methods.

Classes that implement interfaces can also be used as base classes. (The first example above declares *TMemoryManager* as a direct descendent of *TInterfacedObject*.) Since every interface inherits from *IInterface*, a class that implements interfaces must implement the *QueryInterface*, *_AddRef*, and *_Release* methods. The *System* unit's *TInterfacedObject* implements these methods and is thus a convenient base from which to derive other classes that implement interfaces.

When an interface is implemented, each of its methods is mapped onto a method in the implementing class that has the same result type, the same calling convention, the same number of parameters, and identically typed parameters in each position. By default, each interface method is mapped to a method of the same name in the implementing class.

Method resolution clauses

You can override the default name-based mappings by including *method resolution clauses* in a class declaration. When a class implements two or more interfaces that have identically named methods, use method resolution clauses to resolve the naming conflicts.

A method resolution clause has the form

`procedure` *interface*`.`*interfaceMethod* `=` *implementingMethod*`;`

or

`function` *interface*`.`*interfaceMethod* `=` *implementingMethod*`;`

where *implementingMethod* is a method declared in the class or one of its ancestors. The *implementingMethod* can be a method declared later in the class declaration, but cannot be a private method of an ancestor class declared in another module.

For example, the class declaration

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
    ⋮
  end;
```

maps *IMalloc*'s *Alloc* and *Free* methods onto *TMemoryManager*'s *Allocate* and *Deallocate* methods.

A method resolution clause cannot alter a mapping introduced by an ancestor class.

Changing inherited implementations

Descendant classes can change the way a specific interface method is implemented by overriding the implementing method. This requires that the implementing method be virtual or dynamic.

A class can also reimplement an entire interface that it inherits from an ancestor class. This involves relisting the interface in the descendant class's declaration. For example,

```
type
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    ⋮
  end;

  TWindow = class(TInterfacedObject, IWindow)  // TWindow implements IWindow
    procedure Draw;
    ⋮
  end;

  TFrameWindow = class(TWindow, IWindow)  // TFrameWindow reimplements IWindow
    procedure Draw;
    ⋮
  end;
```

Reimplementing an interface hides the inherited implementation of the same interface. Hence method resolution clauses in an ancestor class have no effect on the reimplemented interface.

Implementing interfaces by delegation

The **implements** directive allows you to delegate implementation of an interface to a property in the implementing class. For example,

```
property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
```

declares a property called *MyInterface* that implements the interface *IMyInterface*.

The **implements** directive must be the last specifier in the property declaration and can list more than one interface, separated by commas. The delegate property

- must be of a class or interface type.
- cannot be an array property or have an index specifier.
- must have a **read** specifier. If the property uses a **read** method, that method must use the default **register** calling convention and cannot be dynamic (though it can be virtual) or specify the **message** directive.

Note The class you use to implement the delegated interface should derive from *TAggregatedObject*.

Delegating to an interface-type property

If the delegate property is of an interface type, that interface, or an interface from which it derives, must occur in the ancestor list of the class where the property is declared. The delegate property must return an object whose class completely implements the interface specified by the **implements** directive, and which does so without method resolution clauses. For example,

```
type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;

  TMyClass = class(TObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
  end;

var
  MyClass: TMyClass;
  MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyInterface := ...  // some object whose class implements IMyInterface
  MyInterface := MyClass;
  MyInterface.P1;
end;
```

Delegating to a class-type property

If the delegate property is of a class type, that class and its ancestors are searched for methods implementing the specified interface before the enclosing class and its ancestors are searched. Thus it is possible to implement some methods in the class specified by the property, and others in the class where the property is declared. Method resolution clauses can be used in the usual way to resolve ambiguities or specify a particular method. An interface cannot be implemented by more than one class-type property. For example,

```
type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;
  TMyImplClass = class
    procedure P1;
    procedure P2;
  end;
  TMyClass = class(TInterfacedObject, IMyInterface)
    FMyImplClass: TMyImplClass;
    property MyImplClass: TMyImplClass read FMyImplClass implements IMyInterface;
    procedure IMyInterface.P1 = MyP1;
    procedure MyP1;
  end;
procedure TMyImplClass.P1;
  ⋮
procedure TMyImplClass.P2;
  ⋮
procedure TMyClass.MyP1;
  ⋮
var
  MyClass: TMyClass;
  MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyImplClass := TMyImplClass.Create;
  MyInterface := MyClass;
  MyInterface.P1;       // calls TMyClass.MyP1;
  MyInterface.P2;       // calls TImplClass.P2;
end;
```

Interface references

If you declare a variable of an interface type, the variable can reference instances of any class that implements the interface. Such variables allow you to call interface methods without knowing at compile time where the interface is implemented. But they are subject to the following limitations.

- An interface-type expression gives you access only to methods and properties declared in the interface, not to other members of the implementing class.
- An interface-type expression cannot reference an object whose class implements a descendant interface, unless the class (or one that it inherits from) explicitly implements the ancestor interface as well.

For example,

```
type
  IAncestor = interface
  end;

  IDescendant = interface(IAncestor)
    procedure P1;
  end;

  TSomething = class(TInterfacedObject, IDescendant)
    procedure P1;
```

```
      procedure P2;
    end;
    ⋮
  var
    D: IDescendant;
    A: IAncestor;
  begin
    D := TSomething.Create;  // works!
    A := TSomething.Create;  // error
    D.P1;  // works!
    D.P2;  // error
  end;
```

In this example,

- *A* is declared as a variable of type *IAncestor*. Because *TSomething* does not list *IAncestor* among the interfaces it implements, a *TSomething* instance cannot be assigned to *A*. But if we changed *TSomething*'s declaration to

  ```
  TSomething = class(TInterfacedObject, IAncestor, IDescendant)
    ⋮
  ```

  the first error would become a valid assignment.
- *D* is declared as a variable of type *IDescendant*. While *D* references an instance of *TSomething*, we cannot use it to access *TSomething*'s *P2* method, since *P2* is not a method of *IDescendant*. But if we changed *D*'s declaration to

  ```
  D: TSomething;
  ```

  the second error would become a valid method call.

Interface references are managed through reference-counting, which depends on the *_AddRef* and *_Release* methods inherited from *IInterface*. When an object is referenced only through interfaces, there is no need to destroy it manually; the object is automatically destroyed when the last reference to it goes out of scope.

Global interface-type variables can be initialized only to **nil**.

To determine whether an interface-type expression references an object, pass it to the standard function *Assigned*.

Interface assignment-compatibility

A class type is assignment-compatible with any interface type implemented by the class. An interface type is assignment-compatible with any ancestor interface type. The value **nil** can be assigned to any interface-type variable.

An interface-type expression can be assigned to a variant. If the interface is of type *IDispatch* or a descendant, the variant receives the type code *varDispatch*. Otherwise, the variant receives the type code *varUnknown*.

A variant whose type code is *varEmpty*, *varUnknown*, or *varDispatch* can be assigned to an *IInterface* variable. A variant whose type code is *varEmpty* or *varDispatch* can be assigned to an *IDispatch* variable.

Interface typecasts

Interface types follow the same rules as class types in variable and value typecasts. Class-type expressions can be cast to interface types—for example, `IMyInterface(SomeObject)`—provided the class implements the interface.

An interface-type expression can be cast to *Variant*. If the interface is of type *IDispatch* or a descendant, the resulting variant has the type code *varDispatch*. Otherwise, the resulting variant has the type code *varUnknown*.

A variant whose type code is *varEmpty*, *varUnknown*, or *varDispatch* can be cast to *IInterface*. A variant whose type code is *varEmpty* or *varDispatch* can be cast to *IDispatch*.

Interface querying

You can use the **as** operator to perform checked interface typecasts. This is known as *interface querying*, and it yields an interface-type expression from an object reference or from another interface reference, based on the actual (runtime) type of the object. An interface query has the form

object `as` *interface*

where *object* is an expression of an interface or variant type or denotes an instance of a class that implements an interface, and *interface* is any interface declared with a GUID.

An interface query returns **nil** if *object* is **nil**. Otherwise, it passes the GUID of *interface* to the *QueryInterface* method in *object*, raising an exception unless *QueryInterface* returns zero. If *QueryInterface* returns zero (indicating that *object*'s class implements *interface*), the interface query returns an interface reference to *object*.

Automation objects (Windows only)

An object whose class implements the *IDispatch* interface (declared in the *System* unit) is an Automation object. Automation is available on Windows only.

Dispatch interface types (Windows only)

Dispatch interface types define the methods and properties that an Automation object implements through *IDispatch*. Calls to methods of a dispatch interface are routed through *IDispatch*'s *Invoke* method at runtime; a class cannot implement a dispatch interface.

A dispatch interface type declaration has the form

```
type interfaceName = dispinterface
  ['{GUID}']
  memberList
end;
```

where `['{`*GUID*`}']` is optional and *memberList* consists of property and method declarations. Dispatch interface declarations are similar to regular interface declarations, but they cannot specify an ancestor. For example,

```
type
  IStringsDisp = dispinterface
    ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
    property ControlDefault[Index: Integer]: OleVariant dispid 0; default;
    function Count: Integer; dispid 1;
    property Item[Index: Integer]: OleVariant dispid 2;
    procedure Remove(Index: Integer); dispid 3;
    procedure Clear; dispid 4;
    function Add(Item: OleVariant): Integer; dispid 5;
    function _NewEnum: IUnknown; dispid -4;
  end;
```

Dispatch interface methods (Windows only)

Methods of a dispatch interface are prototypes for calls to the *Invoke* method of the underlying *IDispatch* implementation. To specify an Automation dispatch ID for a method, include the **dispid** directive in its declaration, followed by an integer constant; specifying an already used ID causes an error.

A method declared in a dispatch interface cannot contain directives other than **dispid**. Parameter and result types must be automatable—that is, they must be *Byte*, *Currency*, *Real*, *Double*, *Longint*, *Integer*, *Single*, *Smallint*, *AnsiString*, *WideString*, *TDateTime*, *Variant*, *OleVariant*, *WordBool*, or any interface type.

Dispatch interface properties

Properties of a dispatch interface do not include access specifiers. They can be declared as **read only** or **write only**. To specify a dispatch ID for a property, include the **dispid** directive in its declaration, followed by an integer constant; specifying an already used ID causes an error. Array properties can be declared as **default**. No other directives are allowed in dispatch-interface property declarations.

Accessing Automation objects (Windows only)

Use variants to access Automation objects. When a variant references an Automation object, you can call the object's methods and read or write to its properties through the variant. To do this, you must include *ComObj* in the **uses** clause of one of your units or your program or library.

Automation object method calls are bound at runtime and require no previous method declarations. The validity of these calls is not checked at compile time.

The following example illustrates Automation method calls. The *CreateOleObject* function (defined in *ComObj*) returns an *IDispatch* reference to an Automation object and is assignment-compatible with the variant *Word*.

```
var
  Word: Variant;
begin
  Word := CreateOleObject('Word.Basic');
  Word.FileNew('Normal');
  Word.Insert('This is the first line'#13);
  Word.Insert('This is the second line'#13);
  Word.FileSaveAs('c:\temp\test.txt', 3);
end;
```

You can pass interface-type parameters to Automation methods.

Variant arrays with an element type of *varByte* are the preferred method of passing binary data between Automation controllers and servers. Such arrays are subject to no translation of their data, and can be efficiently accessed using the *VarArrayLock* and *VarArrayUnlock* routines.

Automation object method-call syntax

The syntax of an Automation object method call or property access is similar to that of a normal method call or property access. Automation method calls, however, can use both *positional* and *named* parameters. (But some Automation servers do not support named parameters.)

A positional parameter is simply an expression. A named parameter consists of a parameter identifier, followed by the **:=** symbol, followed by an expression. Positional parameters must precede any named parameters in a method call. Named parameters can be specified in any order.

Some Automation servers allow you to omit parameters from a method call, accepting their default values. For example,

```
Word.FileSaveAs('test.doc');
Word.FileSaveAs('test.doc', 6);
Word.FileSaveAs('test.doc',,,'secret');
Word.FileSaveAs('test.doc', Password := 'secret');
Word.FileSaveAs(Password := 'secret', Name := 'test.doc');
```

Automation method call parameters can be of integer, real, string, Boolean, and variant types. A parameter is passed by reference if the parameter expression consists only of a variable reference, and if the variable reference is of type *Byte*, *Smallint*, *Integer*, *Single*, *Double*, *Currency*, *TDateTime*, *AnsiString*, *WordBool*, or *Variant*. If the expression is not of one of these types, or if it is not just a variable, the parameter is passed by value. Passing a parameter by reference to a method that expects a value parameter causes COM to fetch the value from the reference parameter. Passing a parameter by value to a method that expects a reference parameter causes an error.

Dual interfaces (Windows only)

A dual interface is an interface that supports both compile-time binding and runtime binding through Automation. Dual interfaces must descend from *IDispatch*.

All methods of a dual interface (except from those inherited from *IInterface* and *IDispatch*) must use the **safecall** convention, and all method parameter and result types must be automatable. (The automatable types are *Byte*, *Currency*, *Real*, *Double*, *Real48*, *Integer*, *Single*, *Smallint*, *AnsiString*, *ShortString*, *TDateTime*, *Variant*, *OleVariant*, and *WordBool*.)

Chapter

11

Memory management

This chapter explains how programs use memory and describes the internal formats of Object Pascal data types.

The memory manager (Windows only)

Note Linux uses glibc functions such as *malloc* for memory management. For information, refer to the *malloc* man page on your Linux system.

On Windows systems, the memory manager manages all dynamic memory allocations and deallocations in an application. The *New*, *Dispose*, *GetMem*, *ReallocMem*, and *FreeMem* standard procedures use the memory manager, and all objects and long strings are allocated through the memory manager.

On Windows, the memory manager is optimized for applications that allocate large numbers of small- to medium-sized blocks, as is typical for object-oriented applications and applications that process string data. Other memory managers, such as the implementations of *GlobalAlloc*, *LocalAlloc*, and private heap support in Windows, typically do not perform well in such situations, and would slow down an application if they were used directly.

To ensure the best performance, the memory manager interfaces directly with the Win32 virtual memory API (the *VirtualAlloc* and *VirtualFree* functions). The memory manager reserves memory from the operating system in 1-MB sections of address space, and commits memory as required in 16-KB increments. It decommits and releases unused memory in 16-KB and 1-MB sections. For smaller blocks, committed memory is further suballocated.

Memory manager blocks are always rounded upward to a 4-byte boundary, and always include a 4-byte header in which the size of the block and other status bits are stored. This means that memory manager blocks are always double-word-aligned, which guarantees optimal CPU performance when addressing the block.

The memory manager maintains two status variables, *AllocMemCount* and *AllocMemSize*, which contain the number of currently allocated memory blocks and the combined size of all currently allocated memory blocks. Applications can use these variables to display status information for debugging.

The *System* unit provides two procedures, *GetMemoryManager* and *SetMemoryManager*, that allow applications to intercept low-level memory manager calls. The *System* unit also provides a function called *GetHeapStatus* that returns a record containing detailed memory-manager status information. For further information about these routines, see the online Help.

Variables

Global variables are allocated on the application data segment and persist for the duration of the program. Local variables (declared within procedures and functions) reside in an application's stack. Each time a procedure or function is called, it allocates a set of local variables; on exit, the local variables are disposed of. Compiler optimization may eliminate variables earlier.

Note On Linux, stack size is set by the environment only.

On Windows, an application's stack is defined by two values: the *minimum stack size* and the *maximum stack size*. The values are controlled through the **$MINSTACKSIZE** and **$MAXSTACKSIZE** compiler directives, and default to 16,384 (16K) and 1,048,576 (1M) respectively. An application is guaranteed to have the minimum stack size available, and an application's stack is never allowed to grow larger than the maximum stack size. If there is not enough memory available to satisfy an application's minimum stack requirement, Windows will report an error upon attempting to start the application.

If a Windows application requires more stack space than specified by the minimum stack size, additional memory is automatically allocated in 4K increments. If allocation of additional stack space fails, either because more memory is not available or because the total size of the stack would exceed the maximum stack size, an *EStackOverflow* exception is raised. (Stack overflow checking is completely automatic. The **$S** compiler directive, which originally controlled overflow checking, is maintained for backward compatibility.)

On Windows or Linux, dynamic variables created with the *GetMem* or *New* procedure are heap-allocated and persist until they are deallocated with *FreeMem* or *Dispose*.

Long strings, wide strings, dynamic arrays, variants, and interfaces are heap-allocated, but their memory is managed automatically.

Internal data formats

The following sections describe the internal formats of Object Pascal data types.

Integer types

The format of an integer-type variable depends on its minimum and maximum bounds.

- If both bounds are within the range –128..127 (*Shortint*), the variable is stored as a signed byte.
- If both bounds are within the range 0..255 (*Byte*), the variable is stored as an unsigned byte.
- If both bounds are within the range –32768..32767 (*Smallint*), the variable is stored as a signed word.
- If both bounds are within the range 0..65535 (*Word*), the variable is stored as an unsigned word.
- If both bounds are within the range –2147483648..2147483647 (*Longint*), the variable is stored as a signed double word.
- If both bounds are within the range 0..4294967295 (*Longword*), the variable is stored as an unsigned double word.
- Otherwise, the variable is stored as a signed quadruple word (*Int64*).

Character types

A *Char*, an *AnsiChar*, or a subrange of a *Char* type is stored as an unsigned byte. A *WideChar* is stored as an unsigned word.

Boolean types

A *Boolean* type is stored as a *Byte*, a *ByteBool* is stored as a *Byte*, a *WordBool* type is stored as a *Word*, and a *LongBool* is stored as a *Longint*.

A *Boolean* can assume the values 0 (*False*) and 1 (*True*). *ByteBool*, *WordBool*, and *LongBool* types can assume the values 0 (*False*) or nonzero (*True*).

Enumerated types

An enumerated type is stored as an unsigned byte if the enumeration has no more than 256 values and the type was declared in the **{$Z1}** state (the default). If an enumerated type has more than 256 values, or if the type was declared in the **{$Z2}** state, it is stored as an unsigned word. If an enumerated type is declared in the **{$Z4}** state, it is stored as an unsigned double-word.

Real types

The real types store the binary representation of a sign (+ or –), an *exponent*, and a *significand*. A real value has the form

$$+/- \textit{significand} * 2^{\textit{exponent}}$$

where the *significand* has a single bit to the left of the binary decimal point. (That is, $0 <= \textit{significand} < 2$.)

In the figures that follow, the most significant bit is always on the left and the least significant bit on the right. The numbers at the top indicate the width (in bits) of each field, with the leftmost items stored at the highest addresses. For example, for a *Real48* value, *e* is stored in the first byte, *f* in the following five bytes, and *s* in the most significant bit of the last byte.

The Real48 type

A 6-byte (48-bit) *Real48* number is divided into three fields:

1	39	8
s	*f*	*e*

If $0 < e <= 255$, the value v of the number is given by

$$v = (-1)^s * 2^{(e-129)} * (1.f)$$

If $e = 0$, then $v = 0$.

The *Real48* type can't store denormals, NaNs, and infinities. Denormals become zero when stored in a *Real48*, while NaNs and infinities produce an overflow error if an attempt is made to store them in a *Real48*.

The Single type

A 4-byte (32-bit) *Single* number is divided into three fields:

1	8	23
s	*e*	*f*

The value v of the number is given by

if $0 < e < 255$, then $v = (-1)^s * 2^{(e-127)} * (1.f)$

if $e = 0$ and $f <> 0$, then $v = (-1)^s * 2^{(-126)} * (0.f)$
if $e = 0$ and $f = 0$, then $v = (-1)^s * 0$
if $e = 255$ and $f = 0$, then $v = (-1)^s *$ Inf
if $e = 255$ and $f <> 0$, then v is a NaN

The Double type

An 8-byte (64-bit) *Double* number is divided into three fields:

1	11	52
s	*e*	*f*

The value v of the number is given by

if $0 < e < 2047$, then $v = (-1)^s * 2^{(e-1023)} * (1.f)$

if $e = 0$ and $f <> 0$, then $v = (-1)^s * 2^{(-1022)} * (0.f)$
if $e = 0$ and $f = 0$, then $v = (-1)^s * 0$
if $e = 2047$ and $f = 0$, then $v = (-1)^s * \mathrm{Inf}$
if $e = 2047$ and $f <> 0$, then v is a NaN

The Extended type

A 10-byte (80-bit) *Extended* number is divided into four fields:

1	15	1	63
s	*e*	i	*f*

The value v of the number is given by

if $0 <= e < 32767$, then $v = (-1)^s * 2^{(e-16383)} * (i.f)$

if $e = 32767$ and $f = 0$, then $v = (-1)^s * \mathrm{Inf}$
if $e = 32767$ and $f <> 0$, then v is a NaN

The Comp type

An 8-byte (64-bit) *Comp* number is stored as a signed 64-bit integer.

The Currency type

An 8-byte (64-bit) *Currency* number is stored as a scaled and signed 64-bit integer with the four least-significant digits implicitly representing four decimal places.

Pointer types

A *Pointer* type is stored in 4 bytes as a 32-bit address. The pointer value **nil** is stored as zero.

Short string types

A string occupies as many bytes as its maximum length plus one. The first byte contains the current dynamic length of the string, and the following bytes contain the characters of the string.

The length byte and the characters are considered unsigned values. Maximum string length is 255 characters plus a length byte (**string**[255]).

Long string types

A long string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a long string variable is empty (contains a zero-length string), the string pointer is **nil** and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a long-string memory block.

Table 11.1 Long string dynamic memory layout

Offset	Contents
–8	32-bit reference-count
–4	length in bytes
0..*Length* – 1	character string
Length	NULL character

The NULL character at the end of a long string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a long string directly to a null-terminated string.

For string constants and literals, the compiler generates a memory block with the same layout as a dynamically allocated string, but with a reference count of –1. When a long string variable is assigned a string constant, the string pointer is assigned the address of the memory block generated for the string constant. The built-in string handling routines know not to attempt to modify blocks that have a reference count of –1.

Wide string types

Note On Linux, wide strings are implemented exactly as long strings.

On Windows, a wide string variable occupies four bytes of memory which contain a pointer to a dynamically allocated string. When a wide string variable is empty (contains a zero-length string), the string pointer is **nil** and no dynamic memory is associated with the string variable. For a nonempty string value, the string pointer points to a dynamically allocated block of memory that contains the string value in addition to a 32-bit length indicator. The table below shows the layout of a wide string memory block on Windows.

Table 11.2 Wide string dynamic memory layout (Windows only)

Offset	Contents
–4	32-bit length indicator (in bytes)
0..*Length* – 1	character string
Length	NULL character

The string length is the number of bytes, so it is twice the number of wide characters contained in the string.

The NULL character at the end of a wide string memory block is automatically maintained by the compiler and the built-in string handling routines. This makes it possible to typecast a wide string directly to a null-terminated string.

Set types

A set is a bit array where each bit indicates whether an element is in the set or not. The maximum number of elements in a set is 256, so a set never occupies more than 32 bytes. The number of bytes occupied by a particular set is equal to

$(Max \text{ div } 8) – (Min \text{ div } 8) + 1$

where *Max* and *Min* are the upper and lower bounds of the base type of the set. The byte number of a specific element *E* is

$(E \text{ div } 8) – (Min \text{ div } 8)$

and the bit number within that byte is

$E \text{ mod } 8$

where *E* denotes the ordinal value of the element. When possible, the compiler stores sets in CPU registers, but a set always resides in memory if it is larger than the generic *Integer* type or if the program contains code that takes the address of the set.

Static array types

A static array is stored as a contiguous sequence of variables of the component type of the array. The components with the lowest indexes are stored at the lowest memory addresses. A multidimensional array is stored with the rightmost dimension increasing first.

Dynamic array types

A dynamic-array variable occupies four bytes of memory which contain a pointer to the dynamically allocated array. When the variable is empty (uninitialized) or holds a zero-length array, the pointer is **nil** and no dynamic memory is associated with the variable. For a nonempty array, the variable points to a dynamically allocated block of memory that contains the array in addition to a 32-bit length indicator and a 32-bit reference count. The table below shows the layout of a dynamic-array memory block.

Table 11.3 Dynamic array memory layout

Offset	Contents
–8	32-bit reference-count
–4	32-bit length indicator (number of elements)
0..*Length* * (size of element) – 1	array elements

Record types

When a record type is declared in the **{$A+}** state (the default), and when the declaration does not include a **packed** modifier, the type is an *unpacked record type*, and the fields of the record are aligned for efficient access by the CPU. The alignment is controlled by the type of each field and by whether fields are declared together. Every data type has an inherent alignment, which is automatically computed by the compiler. The alignment can be 1, 2, 4, or 8, and represents the byte boundary that a value of the type must be stored on to provide the most efficient access. The table below lists the alignments for all data types.

Table 11.4 Type alignment masks

Type	Alignment
Ordinal types	size of the type (1, 2, 4, or 8)
Real types	2 for *Real48*, 4 for *Single*, 8 for *Double* and *Extended*
Short string types	1
Array types	same as the element type of the array.
Record types	the largest alignment of the fields in the record
Set types	size of the type if 1, 2, or 4, otherwise 1
All other types	determined by the $A directive.

To ensure proper alignment of the fields in an unpacked record type, the compiler inserts an unused byte before fields with an alignment of 2, and up to three unused bytes before fields with an alignment of 4, if required. Finally, the compiler rounds the total size of the record upward to the byte boundary specified by the largest alignment of any of the fields.

If two fields share a common type specification, they are packed even if the declaration does not include the **packed** modifier and the record type is not declared in the **{$A–}** state. Thus, for example, given the following declaration

```
type
  TMyRecord = record
    A, B: Extended;
    C: Extended;
  end;
```

A and B are packed (aligned on byte boundaries) because they share the same type specification. The compiler pads the structure with unused bytes to ensure that C appears on a quadword boundary.

When a record type is declared in the **{$A–}** state, or when the declaration includes the **packed** modifier, the fields of the record are not aligned, but are instead assigned consecutive offsets. The total size of such a packed record is simply the size of all the fields. Because data alignment can change, it's a good idea to pack any record structure that you intend to write to disk or pass in memory to another module compiled using a different version of the compiler.

File types

File types are represented as records. Typed files and untyped files occupy 332 bytes, which are laid out as follows:

```
type
  TFileRec = packed record
    Handle: Integer;
    Mode: word;
    Flags: word;
    case Byte of
      0: (RecSize: Cardinal);
      1: (BufSize: Cardinal;
          BufPos: Cardinal;
          BufEnd: Cardinal;
          BufPtr: PChar;
          OpenFunc: Pointer;
          InOutFunc: Pointer;
          FlushFunc: Pointer;
          CloseFunc: Pointer;
          UserData: array[1..32] of Byte;
          Name: array[0..259] of Char; );
  end;
```

Text files occupy 460 bytes, which are laid out as follows:

```
type
  TTextBuf = array[0..127] of Char;
  TTextRec = packed record
    Handle: Integer;
    Mode: word;
    Flags: word;
    BufSize: Cardinal;
    BufPos: Cardinal;
    BufEnd: Cardinal;
    BufPtr: PChar;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData: array[1..32] of Byte;
    Name: array[0..259] of Char;
    Buffer: TTextBuf;
  end;
```

Handle contains the file's handle (when the file is open).

The *Mode* field can assume one of the values

```
const
  fmClosed = $D7B0;
  fmInput  = $D7B1;
  fmOutput = $D7B2;
  fmInOut  = $D7B3;
```

where *fmClosed* indicates that the file is closed, *fmInput* and *fmOutput* indicate a text file that has been reset (*fmInput*) or rewritten (*fmOutput*), *fmInOut* indicates a typed or untyped file that has been reset or rewritten. Any other value indicates that the file variable is not assigned (and hence not initialized).

The *UserData* field is available for user-written routines to store data in.

Name contains the file name, which is a sequence of characters terminated by a null character (#0).

For typed files and untyped files, *RecSize* contains the record length in bytes, and the *Private* field is unused but reserved.

For text files, *BufPtr* is a pointer to a buffer of *BufSize* bytes, *BufPos* is the index of the next character in the buffer to read or write, and *BufEnd* is a count of valid characters in the buffer. *OpenFunc*, *InOutFunc*, *FlushFunc*, and *CloseFunc* are pointers to the I/O routines that control the file; see “Device functions” on page 8-5. Flags determines the line break style as follows:

bit 0 clear	LF line breaks
bit 0 set	CRLF line breaks

All other Flags bits are reserved for future use. See also *DefaultTextLineBreakStyle* and *SetLineBreakStyle*.

Procedural types

A procedure pointer is stored as a 32-bit pointer to the entry point of a procedure or function. A method pointer is stored as a 32-bit pointer to the entry point of a method, followed by a 32-bit pointer to an object.

Class types

A class-type value is stored as a 32-bit pointer to an instance of the class, which is called an *object*. The internal data format of an object resembles that of a record. The object’s fields are stored in order of declaration as a sequence of contiguous variables. Fields are always aligned, corresponding to an unpacked record type. Any fields inherited from an ancestor class are stored before the new fields defined in the descendant class.

The first 4-byte field of every object is a pointer to the *virtual method table* (VMT) of the class. There is exactly one VMT per class (not one per object); distinct class types, no matter how similar, never share a VMT. VMT’s are built automatically by the compiler, and are never directly manipulated by a program. Pointers to VMT’s, which are automatically stored by constructor methods in the objects they create, are also never directly manipulated by a program.

The layout of a VMT is shown in the following table. At positive offsets, a VMT consists of a list of 32-bit method pointers—one per user-defined virtual method in the class type—in order of declaration. Each slot contains the address of the corresponding virtual method's entry point. This layout is compatible with a C++ v-table and with COM. At negative offsets, a VMT contains a number of fields that are internal to Object Pascal's implementation. Applications should use the methods defined in *TObject* to query this information, since the layout is likely to change in future implementations of Object Pascal.

Table 11.5 Virtual method table layout

Offset	Type	Description
–76	*Pointer*	pointer to virtual method table (or **nil**)
–72	*Pointer*	pointer to interface table (or **nil**)
–68	*Pointer*	pointer to Automation information table (or **nil**)
–64	*Pointer*	pointer to instance initialization table (or **nil**)
–60	*Pointer*	pointer to type information table (or **nil**)
–56	*Pointer*	pointer to field definition table (or **nil**)
–52	*Pointer*	pointer to method definition table (or **nil**)
–48	*Pointer*	pointer to dynamic method table (or **nil**)
–44	*Pointer*	pointer to short string containing class name
–40	*Cardinal*	instance size in bytes
–36	*Pointer*	pointer to a pointer to ancestor class (or **nil**)
–32	*Pointer*	pointer to entry point of *SafecallException* method (or **nil**)
–28	*Pointer*	entry point of *AfterConstruction* method
–24	*Pointer*	entry point of *BeforeDestruction* method
–20	*Pointer*	entry point of *Dispatch* method
–16	*Pointer*	entry point of *DefaultHandler* method
–12	*Pointer*	entry point of *NewInstance* method
–8	*Pointer*	entry point of *FreeInstance* method
–4	*Pointer*	entry point of *Destroy* destructor
0	*Pointer*	entry point of first user-defined virtual method
4	*Pointer*	entry point of second user-defined virtual method
⋮	⋮	⋮

Class reference types

A class-reference value is stored as a 32-bit pointer to the virtual method table (VMT) of a class.

Variant types

A variant is stored as a 16-byte record that contains a type code and a value (or a reference to a value) of the type given by the code. The *System* and *Variants* units define constants and types for variants.

The *TVarData* type represents the internal structure of a *Variant* variable (on Windows, this is identical to the *Variant* type used by COM and the Win32 API). The

TVarData type can be used in typecasts of *Variant* variables to access the internal structure of a variable.

The *VType* field of a *TVarData* record contains the type code of the variant in the lower twelve bits (the bits defined by the *varTypeMask* constant). In addition, the *varArray* bit may be set to indicate that the variant is an array, and the *varByRef* bit may be set to indicate that the variant contains a reference as opposed to a value.

The *Reserved1*, *Reserved2*, and *Reserved3* fields of a *TVarData* record are unused.

The contents of the remaining eight bytes of a *TVarData* record depend on the *VType* field. If neither the *varArray* nor the *varByRef* bits are set, the variant contains a value of the given type.

If the *varArray* bit is set, the variant contains a pointer to a *TVarArray* structure that defines an array. The type of each array element is given by the *varTypeMask* bits in the *VType* field.

If the *varByRef* bit is set, the variant contains a reference to a value of the type given by the *varTypeMask* and *varArray* bits in the *VType* field.

The *varString* type code is private. Variants containing a *varString* value should never be passed to a non-Delphi function. On Windows, Delphi's Automation support automatically converts *varString* variants to *varOleStr* variants before passing them as parameters to external functions.

On Linux, *VT_decimal* is not supported.

Chapter

12

Program control

This chapter explains how parameters and function results are stored and transferred. The final section discusses exit procedures.

Parameters and function results

Treatment of parameters and function results is determined by several factors, including calling conventions, parameter semantics, and the type and size of the value being passed.

Parameter passing

Parameters are transferred to procedures and functions via CPU registers or the stack, depending on the routine's calling convention. For information about calling conventions, see "Calling conventions" on page 6-4.

Variable (**var**) parameters are always passed by reference, as 32-bit pointers that point to the actual storage location.

Value and constant (**const**) parameters are passed by value or by reference, depending on the type and size of the parameter:

- An ordinal parameter is passed as an 8-bit, 16-bit, 32-bit, or 64-bit value, using the same format as a variable of the corresponding type.
- A real parameter is always passed on the stack. A *Single* parameter occupies 4 bytes, and a *Double*, *Comp*, or *Currency* parameter occupies 8 bytes. A *Real48* occupies 8 bytes, with the *Real48* value stored in the lower 6 bytes. An *Extended* occupies 12 bytes, with the *Extended* value stored in the lower 10 bytes.
- A short-string parameter is passed as a 32-bit pointer to a short string.

- A long-string or dynamic-array parameter is passed as a 32-bit pointer to the dynamic memory block allocated for the long string. The value **nil** is passed for an empty long string.
- A pointer, class, class-reference, or procedure-pointer parameter is passed as a 32-bit pointer.
- A method pointer is passed on the stack as two 32-bit pointers. The instance pointer is pushed before the method pointer so that the method pointer occupies the lowest address.
- Under the **register** and **pascal** conventions, a variant parameter is passed as a 32-bit pointer to a *Variant* value.
- Sets, records, and static arrays of 1, 2, or 4 bytes are passed as 8-bit, 16-bit, and 32-bit values. Larger sets, records, and static arrays are passed as 32-bit pointers to the value. An exception to this rule is that records are always passed directly on the stack under the **cdecl**, **stdcall**, and **safecall** conventions; the size of a record passed this way is rounded upward to the nearest double-word boundary.
- An open-array parameter is passed as two 32-bit values. The first value is a pointer to the array data, and the second value is one less than the number of elements in the array.

When two parameters are passed on the stack, each parameter occupies a multiple of 4 bytes (a whole number of double words). For an 8-bit or 16-bit parameter, even though the parameter occupies only a byte or a word, it is passed as a double word. The contents of the unused parts of the double word are undefined.

Under the **pascal**, **cdecl**, **stdcall** and **safecall** conventions, all parameters are passed on the stack. Under the **pascal** convention, parameters are pushed in the order of their declaration (left-to-right), so that the first parameter ends up at the highest address and the last parameter ends up at the lowest address. Under the **cdecl**, **stdcall**, and **safecall** conventions, parameters are pushed in reverse order of declaration (right-to-left), so that the first parameter ends up at the lowest address and the last parameter ends up at the highest address.

Under the **register** convention, up to three parameters are passed in CPU registers, and the rest (if any) are passed on the stack. The parameters are passed in order of declaration (as with the **pascal** convention), and the first three parameters that qualify are passed in the EAX, EDX, and ECX registers, in that order. Real, method-pointer, variant, *Int64*, and structured types (see "Structured types" on page 5-16 do not qualify as register parameters, but all other parameters do. If more than three parameters qualify as register parameters, the first three are passed in EAX, EDX, and ECX, and the remaining parameters are pushed onto the stack in order of declaration. For example, given the declaration

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

a call to *Test* passes *A* in EAX as a 32-bit integer, *B* in EDX as a pointer to a *Char*, and *D* in ECX as a pointer to a long-string memory block; *C* and *E* are pushed onto the stack as two double-words and a 32-bit pointer, in that order.

Register saving conventions

Procedures and functions must preserve the EBX, ESI, EDI, and EBP registers, but can modify the EAX, EDX, and ECX registers. When implementing a constructor or destructor in assembler, be sure to preserve the DL register. Procedures and functions are invoked with the assumption that the CPU's direction flag is cleared (corresponding to a CLD instruction) and must return with the direction flag cleared.

Note Object Pascal procedures and functions are generally invoked with the assumption that the FPU stack is empty: The compiler tries to use all eight FPU stack entries when it generates code.

When working with the MMX and XMM instructions, be user to preserve the values of the xmm and mm registers. Object Pascal functions are invoked with the assumption that the x87 FPU data registers are are available for use by x87 floating point instructions. That is, the compiler assumes that the EMMS/FEMMS has been called after MMX operations. Object Pascal functions do not make any assumptions about the state and content of xmm registers. They do not guarantee that the content of xmm registers is unchanged.

Function results

The following conventions are used for returning function result values.

- Ordinal results are returned, when possible, in a CPU register. Bytes are returned in AL, words are returned in AX, and double-words are returned in EAX.
- Real results are returned in the floating-point coprocessor's top-of-stack register (ST(0)). For function results of type *Currency*, the value in ST(0) is scaled by 10000. For example, the *Currency* value 1.234 is returned in ST(0) as 12340.
- For a string, dynamic array, method pointer, variant, or *Int64* result, the effects are the same as if the function result were declared as an additional **var** parameter following the declared parameters. In other words, the caller passes an additional 32-bit pointer that points to a variable in which to return the function result.
- Pointer, class, class-reference, and procedure-pointer results are returned in EAX.
- For static-array, record, and set results, if the value occupies one byte it is returned in AL; if the value occupies two bytes it is returned in AX; and if the value occupies four bytes it is returned in EAX. Otherwise, the result is returned in an additional **var** parameter that is passed to the function after the declared parameters.

Method calls

Methods use the same calling conventions as ordinary procedures and functions, except that every method has an additional implicit parameter *Self*, which is a reference to the instance or class in which the method is called. The *Self* parameter is passed as a 32-bit pointer.

- Under the **register** convention, *Self* behaves as if it were declared *before* all other parameters. It is therefore always passed in the EAX register.
- Under the **pascal** convention, *Self* behaves as if it were declared *after* all other parameters (including the additional **var** parameter sometimes passed for a function result). It is therefore pushed last, ending up at a lower address than all other parameters.
- Under the **cdecl**, **stdcall**, and **safecall** conventions, *Self* behaves as if it were declared *before* all other parameters, but *after* the additional **var** parameter (if any) passed for a function result. It is therefore the last to be pushed, except for the additional **var** parameter.

Constructors and destructors

Constructors and destructors use the same calling conventions as other methods, except that an additional *Boolean* flag parameter is passed to indicate the context of the constructor or destructor call.

A value of *False* in the flag parameter of a constructor call indicates that the constructor was invoked through an instance object or using the **inherited** keyword. In this case, the constructor behaves like an ordinary method. A value of *True* in the flag parameter of a constructor call indicates that the constructor was invoked through a class reference. In this case, the constructor creates an instance of the class given by *Self*, and returns a reference to the newly created object in EAX.

A value of *False* in the flag parameter of a destructor call indicates that the destructor was invoked using the **inherited** keyword. In this case, the destructor behaves like an ordinary method. A value of *True* in the flag parameter of a destructor call indicates that the destructor was invoked through an instance object. In this case, the destructor deallocates the instance given by *Self* just before returning.

The flag parameter behaves as if it were declared before all other parameters. Under the **register** convention, it is passed in the DL register. Under the **pascal** convention, it is pushed before all other parameters. Under the **cdecl**, **stdcall**, and **safecall** conventions, it is pushed just before the *Self* parameter.

Since the DL register indicates whether the constructor or destructor is the outermost in the call stack, you must restore the value of DL before exiting so that *BeforeDestruction* or *AfterConstruction* can be called properly.

Exit procedures

Exit procedures ensure that specific actions—such as updating and closing files—are carried out before a program terminates. The *ExitProc* pointer variable allows you to “install” an exit procedure, so that it is always called as part of the program’s termination—whether the termination is normal, forced by a call to *Halt*, or the result of a runtime error. An exit procedure takes no parameters.

Note It is recommended that you use finalization sections rather than exit procedures for all exit behavior. (See “The finalization section” on page 3-5.) Exit procedures are available only for executables. For shared objects (Linux) or .DLLs (Windows) you can use a similar variable, *DllProc*, which is called when the library is loaded as well as when it is unloaded. For packages, exit behavior must be implemented in a finalization section. All exit procedures are called before execution of finalization sections.

Units as well as programs can install exit procedures. A unit can install an exit procedure as part of its initialization code, relying on the procedure to close files or perform other clean-up tasks.

When implemented properly, an exit procedure is part of a chain of exit procedures. The procedures are executed in reverse order of installation, ensuring that the exit code of one unit isn't executed before the exit code of any units that depend on it. To keep the chain intact, you must save the current contents of *ExitProc* before pointing it to the address of your own exit procedure. Also, the first statement in your exit procedure must reinstall the saved value of *ExitProc*.

The following code shows a skeleton implementation of an exit procedure.

```
var
  ExitSave: Pointer;

procedure MyExit;
begin
  ExitProc := ExitSave;  // always restore old vector first
  ⋮
end;

begin
  ExitSave := ExitProc;
  ExitProc := @MyExit;
  ⋮
end.
```

On entry, the code saves the contents of *ExitProc* in *ExitSave*, then installs the *MyExit* procedure. When called as part of the termination process, the first thing *MyExit* does is reinstall the previous exit procedure.

The termination routine in the runtime library keeps calling exit procedures until *ExitProc* becomes **nil**. To avoid infinite loops, *ExitProc* is set to **nil** before every call, so the next exit procedure is called only if the current exit procedure assigns an address to *ExitProc*. If an error occurs in an exit procedure, it is not called again.

An exit procedure can learn the cause of termination by examining the *ExitCode* integer variable and the *ErrorAddr* pointer variable. In case of normal termination, *ExitCode* is zero and *ErrorAddr* is **nil**. In case of termination through a call to *Halt*, *ExitCode* contains the value passed to *Halt* and *ErrorAddr* is **nil**. In case of termination due to a runtime error, *ExitCode* contains the error code and *ErrorAddr* contains the address of the invalid statement.

The last exit procedure (the one installed by the runtime library) closes the *Input* and *Output* files. If *ErrorAddr* is not **nil**, it outputs a runtime error message. To output your own runtime error message, install an exit procedure that examines *ErrorAddr* and outputs a message if it's not **nil**; before returning, set *ErrorAddr* to **nil** so that the error is not reported again by other exit procedures.

Once the runtime library has called all exit procedures, it returns to the operating system, passing the value stored in *ExitCode* as a return code.

Chapter 13

Inline assembly code

The built-in assembler allows you to write assembly code within Object Pascal programs. It has the following features:

- Allows for inline assembly
- Supports all instructions found in the Intel Pentium III, Intel MMX extensions, Streaming SIMD Extensions (SSE), and the AMD Athlon (including 3D Now!)
- Provides no macro support, but allows for pure assembly function procedures
- Permits the use of Object Pascal identifiers, such as constants, types, and variables in assembly statements

As an alternative to the built-in assembler, you can link to object files that contain external procedures and functions. See "Linking to object files" on page 6-7 for more information.

Note If you have external assembly code that you want to use in your applications, you should consider rewriting it in Object Pascal or minimally reimplement it using the inline assembler.

The asm statement

The built-in assembler is accessed through **asm** statements, which have the form

`asm` *statementList* `end`

where *statementList* is a sequence of assembly statements separated by semicolons, end-of-line characters, or Object Pascal comments.

Comments in an **asm** statement must be in Object Pascal style. A semicolon does not indicate that the rest of the line is a comment.

The reserved word **inline** and the directive **assembler** are maintained for backward compatibility only. They have no effect on the compiler.

Register use

In general, the rules of register use in an **asm** statement are the same as those of an **external** procedure or function. An **asm** statement must preserve the EDI, ESI, ESP, EBP, and EBX registers, but can freely modify the EAX, ECX, and EDX registers. On entry to an **asm** statement, EBP points to the current stack frame and ESP points to the top of the stack. Except for ESP and EBP, an **asm** statement can assume nothing about register contents on entry to the statement.

Assembler statement syntax

This syntax of an assembly statement is

$Label$: $Prefix$ $Opcode$ $Operand_1$, $Operand_2$

where *Label* is a label, *Prefix* is an assembly prefix opcode (operation code), *Opcode* is an assembly instruction opcode or directive, and *Operand* is an assembly expression. *Label* and *Prefix* are optional. Some opcodes take only one operand, and some take none.

Comments are allowed between assembly statements, but not within them. For example,

```
MOV AX,1 {Initial value}     { OK }
MOV CX,100 {Count}           { OK }

MOV {Initial value} AX,1;    { Error! }
MOV CX, {Count} 100          { Error! }
```

Labels

Labels are used in built-in assembly statements as they are in Object Pascal—by writing the label and a colon before a statement. There is no limit to a label's length. As in Object Pascal, labels must be declared in a **label** declaration part in the block containing the **asm** statement. There is one exception to this rule: *local labels*.

Local labels are labels that start with an at-sign (@). They consist of an at-sign followed by one or more letters, digits, underscores, or at-signs. Use of local labels is restricted to **asm** statements, and the scope of a local label extends from the **asm** reserved word to the end of the **asm** statement that contains it. A local label doesn't have to be declared.

Instruction opcodes

The built-in assembler supports all of the Intel-documented opcodes for general application use. Note that operating system privileged instructions may not be supported. Specifically, the following families of instructions are supported:

- Pentium family
- Pentium Pro and Pentium II

- Pentium III
- Pentium 4

Note Pentium 4 instructions are only supported on Windows.

In addition, the built-in assembler supports the following instruction sets

- AMD 3DNow! (from the AMD K6 onwards)
- AMD Enhanced 3DNow! (from the AMD Athlon onwards)

For a complete description of each instruction, refer to your microprocessor documentation.

RET instruction sizing

The RET instruction opcode always generates a near return.

Automatic jump sizing

Unless otherwise directed, the built-in assembler optimizes jump instructions by automatically selecting the shortest, and therefore most efficient, form of a jump instruction. This automatic jump sizing applies to the unconditional jump instruction (JMP), and to all conditional jump instructions when the target is a label (not a procedure or function).

For an unconditional jump instruction (JMP), the built-in assembler generates a short jump (one-byte opcode followed by a one-byte displacement) if the distance to the target label is –128 to 127 bytes. Otherwise it generates a near jump (one-byte opcode followed by a two-byte displacement).

For a conditional jump instruction, a short jump (one-byte opcode followed by a one-byte displacement) is generated if the distance to the target label is –128 to 127 bytes. Otherwise, the built-in assembler generates a short jump with the inverse condition, which jumps over a near jump to the target label (five bytes in total). For example, the assembly statement

```
JC      Stop
```

where *Stop* isn't within reach of a short jump, is converted to a machine code sequence that corresponds to this:

```
JNC      Skip
JMP      Stop
Skip:
```

Jumps to the entry points of procedures and functions are always near.

Assembly directives

The built-in assembler supports three assembly define directives: DB (define byte), DW (define word), and DD (define double word). Each generates data corresponding to the comma-separated operands that follow the directive.

The DB directive generates a sequence of bytes. Each operand can be a constant expression with a value between –128 and 255, or a character string of any length.

Constant expressions generate one byte of code, and strings generate a sequence of bytes with values corresponding to the ASCII code of each character.

The DW directive generates a sequence of words. Each operand can be a constant expression with a value between –32,768 and 65,535, or an address expression. For an address expression, the built-in assembler generates a near pointer—that is, a word that contains the offset part of the address.

The DD directive generates a sequence of double words. Each operand can be a constant expression with a value between –2,147,483,648 and 4,294,967,295, or an address expression. For an address expression, the built-in assembler generates a far pointer—that is, a word that contains the offset part of the address, followed by a word that contains the segment part of the address.

The DQ directive defines a quad word for Int64 values.

The data generated by the DB, DW, and DD directives is always stored in the code segment, just like the code generated by other built-in assembly statements. To generate uninitialized or initialized data in the data segment, you should use Object Pascal **var** or **const** declarations.

Some examples of DB, DW, and DD directives follow.

```
asm
  DB      0FFH                          { One byte }
  DB      0,99                          { Two bytes }
  DB      'A'                           { Ord('A') }
  DB      'Hello world...',0DH,0AH      { String followed by CR/LF }
  DB      12,"string"                   { Object Pascal style string }
  DW      0FFFFH                        { One word }
  DW      0,9999                        { Two words }
  DW      'A'                           { Same as DB 'A',0 }
  DW      'BA'                          { Same as DB 'A','B' }
  DW      MyVar                         { Offset of MyVar }
  DW      MyProc                        { Offset of MyProc }
  DD      0FFFFFFFFH                    { One double-word }
  DD      0,999999999                   { Two double-words }
  DD      'A'                           { Same as DB 'A',0,0,0 }
  DD      'DCBA'                        { Same as DB 'A','B','C','D' }
  DD      MyVar                         { Pointer to MyVar }
  DD      MyProc                        { Pointer to MyProc }
end;
```

When an identifier precedes a DB, DW, or DD directive, it causes the declaration of a byte-, word-, or double-word-sized variable at the location of the directive. For example, the assembler allows the following:

```
ByteVar     DB      ?
WordVar     DW      ?
IntVar      DD      ?
  ⋮
            MOV     AL,ByteVar
            MOV     BX,WordVar
            MOV     ECX,IntVar
```

The built-in assembler doesn't support such variable declarations. The only kind of symbol that can be defined in an inline assembly statement is a label. All variables must be declared using Object Pascal syntax; the preceding construction can be replaced by

```
var
  ByteVar: Byte;
  WordVar: Word;
  IntVar: Integer;
  ⋮
asm
  MOV     AL,ByteVar
  MOV     BX,WordVar
  MOV     ECX,IntVar
end;
```

SMALL and LARGE can be used determine the width of a displacement:

```
MOV EAX, [LARGE $1234]
```

This instruction generates a "normal" move with a 32-bit displacement ($00001234).

```
MOV EAX, [SMALL $1234]
```

The second instruction will generate a move with an address size override prefix and a 16-bit displacement ($1234).

SMALL can be used to save space. The following example generates an address size override and a 2-byte address (in total three bytes)

```
MOV EAX, [SMALL 123]
```

as opposed to

```
MOV EAX, [123]
```

which will generate no address size override and a 4-byte address (in total four bytes).

Two additional directives allow assembly code to access dynamic and virtual methods: VMTOFFSET and DMTINDEX.

VMTOFFSET retrieves the offset in bytes of the virtual method pointer table entry of the virtual method argument from the beginning of the virtual method table (VMT). This directive needs a fully specified class name with a method name as a parameter, for example, TExample.VirtualMethod.

DMTINDEX retrieves the dynamic method table index of the passed dynamic method. This directive also needs a fully specified class name with a method name as a parameter, for example, TExample.DynamicMethod. To invoke the dynamic method, call System.@CallDynaInst with the (E)SI register containing the value obtained from DMTINDEX.

Note Methods with the "message" directive are implemented as dynamic methods and can also be called using the DMTINDEX technique. For example:

```
TMyClass = class
  procedure x; message MYMESSAGE;
end;
```

The following example uses both DMTINDEX and VMTOFFSET to access dynamic and virtual methods:

```
program Project2;

type
  TExample = class
    procedure DynamicMethod; dynamic;
    procedure VirtualMethod; virtual;
  end;

procedure TExample.DynamicMethod;
begin

end;

procedure TExample.VirtualMethod;
begin

end;

procedure CallDynamicMethod(e: TExample);
asm
  // Save ESI register
  PUSH    ESI

  // Instance pointer needs to be in EAX
  MOV     EAX, e
  // DMT entry index needs to be in (E)SI
  MOV     ESI, DMTINDEX TExample.DynamicMethod
  // Now call the method
  CALL    System.@CallDynaInst

  // Restore ESI register
  POP ESI
end;

procedure CallVirtualMethod(e: TExample);
asm
  // Instance pointer needs to be in EAX
  MOV     EAX, e

  // Retrieve VMT table entry
  MOV     EDX, [EAX]

  // Now call the method at offset VMTOFFSET
  CALL    DWORD PTR [EDX + VMTOFFSET TExample.VirtualMethod]
end;

var
  e: TExample;

begin
  e := TExample.Create;
```

```
    try
      CallDynamicMethod(e);
      CallVirtualMethod(e);
    finally
      e.Free;
    end;
  end.
```

Operands

Inline assembler operands are expressions that consist of constants, registers, symbols, and operators.

Within operands, the following reserved words have predefined meanings:

Table 13.1 Built-in assembler reserved words

AH
AL
AND
AX
BH
BL
BP
BX
BYTE
CH
CL
CS
CX
DH
DI
DL
CL
DMTINDEX
DS
DWORD
DX
EAX
EBP
EBX
ECX
EDI
EDX
EIP
ES
ESI
ESP
FS
GS
HIGH
LARGE
LOW
mm0
mm1
mm2
mm3
mm4
mm5
mm6
mm7
MOD
NOT
OFFSET
OR
PTR
QWORD
SHL
SHR
SI
SMALL
SP
SS
ST
TBYTE
TYPE
VMTOFFSET
WORD
xmm0
xmm1
xmm2
xmm3
xmm4
xmm5
xmm6
xmm7
XOR

Reserved words always take precedence over user-defined identifiers. For example,

```
var
  Ch: Char;
  ⋮
asm
  MOV    CH, 1
end;
```

loads 1 into the CH register, not into the *Ch* variable. To access a user-defined symbol with the same name as a reserved word, you must use the ampersand (&) override operator:

```
MOV    &Ch, 1
```

It is best to avoid user-defined identifiers with the same names as built-in assembler reserved words.

Expressions

The built-in assembler evaluates all expressions as 32-bit integer values. It doesn't support floating-point and string values, except string constants.

Expressions are built from *expression elements* and *operators*, and each expression has an associated *expression class* and *expression type*.

Differences between Object Pascal and assembler expressions

The most important difference between Object Pascal expressions and built-in assembler expressions is that assembler expressions must resolve to a constant value-that is a value that can be computed at compile time. For example, given the declarations

```
const
  X = 10;
  Y = 20;
var
  Z: Integer;
```

the following is a valid statement.

```
asm
  MOV     Z,X+Y
end;
```

Because both *X* and *Y* are constants, the expression `X + Y` is a convenient way of writing the constant 30, and the resulting instruction simply moves of the value 30 into the variable *Z*. But if *X* and *Y* are variables—

```
var
  X, Y: Integer;
```

—the built-in assembler cannot compute the value of `X + Y` at compile time. In this case, to move the sum of *X* and *Y* into *Z* you would use

```
asm
  MOV     EAX,X
  ADD     EAX,Y
  MOV     Z,EAX
end;
```

In an Object Pascal expression, a variable reference denotes the *contents* of the variable. But in an assembler expression, a variable reference denotes the *address* of the variable. In Object Pascal the expression `X + 4` (where *X* is a variable) means the contents of X plus 4, while to the built-in assembler it means the contents of the word at the address four bytes higher than the address of *X*. So, even though you're allowed to write

```
asm
  MOV     EAX,X+4
end;
```

this code doesn't load the value of *X* plus 4 into AX; instead, it loads the value of a word stored four bytes beyond *X*. The correct way to add 4 to the contents of *X* is

```
asm
  MOV     EAX,X
  ADD     EAX,4
end;
```

Expression elements

The elements of an expression are *constants*, *register*s, and *symbols*.

Constants

The built-in assembler supports two types of constant: *numeric constants* and *string constants*.

Numeric constants

Numeric constants must be integers, and their values must be between –2,147,483,648 and 4,294,967,295.

By default, numeric constants use decimal notation, but the built-in assembler also supports binary, octal, and hexadecimal. Binary notation is selected by writing a *B* after the number, octal notation by writing an *O* after the number, and hexadecimal notation by writing an *H* after the number or a **$** before the number.

Numeric constants must start with one of the digits 0 through 9 or the **$** character. When you write a hexadecimal constant using the *H* suffix, an extra zero is required in front of the number if the first significant digit is one of the digits A through F. For example, `0BAD4H` and `$BAD4` are hexadecimal constants, but `BAD4H` is an identifier because it starts with a letter.

String constants

String constants must be enclosed in single or double quotation marks. Two consecutive quotation marks of the same type as the enclosing quotation marks count as only one character. Here are some examples of string constants:

```
'Z'
'Delphi'
‘Linux’
"That's all folks"
'"That''s all folks," he said.'
'100'
'"'
"'"
```

String constants of any length are allowed in DB directives, and cause allocation of a sequence of bytes containing the ASCII values of the characters in the string. In all other cases, a string constant can be no longer than four characters and denotes a numeric value which can participate in an expression. The numeric value of a string constant is calculated as

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

where *Ch1* is the rightmost (last) character and *Ch4* is the leftmost (first) character. If the string is shorter than four characters, the leftmost characters are assumed to be zero. The following table shows string constants and their numeric values.

Table 13.2 String examples and their values

String	Value
`'a'`	00000061H
`'ba'`	00006261H
`'cba'`	00636261H
`'dcba'`	64636261H
`'a '`	00006120H
`'   a'`	20202061H
`'a' * 2`	000000E2H
`'a'-'A'`	00000020H
not `'a'`	FFFFFF9EH

Registers

The following reserved symbols denote CPU registers in the inline assembler:

Table 13.3 CPU registers

32-bit general purpose	EAX EBX ECX EDX	32-bit pointer or index	ESP EBP ESI EDI
16-bit general purpose	AX BX CX DX	16-bit pointer or index	SP BP SI DI
8-bit low registers	AL BL CL DL	16-bit segment registers	CS DS SS ES
		32-bit segment registers	FS GS
8-bit high registers	AH BH CH DH	Coprocessor register stack	ST

When an operand consists solely of a register name, it is called a *register operand*. All registers can be used as register operands, and some registers can be used in other contexts.

The base registers (BX and BP) and the index registers (SI and DI) can be written within square brackets to indicate indexing. Valid base/index register combinations are [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], and [BP+DI]. You can also index with all the 32-bit registers—for example, [EAX+ECX], [ESP], and [ESP+EAX+5].

The segment registers (ES, CS, SS, DS, FS, and GS) are supported, but segments are normally not useful in 32-bit applications.

The symbol ST denotes the topmost register on the 8087 floating-point register stack. Each of the eight floating-point registers can be referred to using ST(*X*), where *X* is a constant between 0 and 7 indicating the distance from the top of the register stack.

Symbols

The built-in assembler allows you to access almost all Object Pascal identifiers in assembly language expressions, including constants, types, variables, procedures, and functions. In addition, the built-in assembler implements the special symbol

@Result, which corresponds to the *Result* variable within the body of a function. For example, the function

```
function Sum(X, Y: Integer): Integer;
begin
  Result := X + Y;
end;
```

could be written in assembly language as

```
function Sum(X, Y: Integer): Integer; stdcall;
begin
  asm
    MOV     EAX,X
    ADD     EAX,Y
    MOV     @Result,EAX
  end;
end;
```

The following symbols cannot be used in **asm** statements:

- Standard procedures and functions (for example, *WriteLn* and *Chr*).
- String, floating-point, and set constants (except when loading registers).
- Labels that aren't declared in the current block.
- The *@Result* symbol outside of functions.

The following table summarizes the kinds of symbol that can be used in **asm** statements.

Table 13.4 Symbols recognized by the built-in assembler

Symbol	Value	Class	Type
Label	Address of label	Memory reference	Size of type
Constant	Value of constant	Immediate value	0
Type	0	Memory reference	Size of type
Field	Offset of field	Memory	Size of type
Variable	Address of variable	Memory reference	Size of type
Procedure	Address of procedure	Memory reference	Size of type
Function	Address of function	Memory reference	Size of type
Unit	0	Immediate value	0
@Result	Result variable offset	Memory reference	Size of type

With optimizations disabled, local variables (variables declared in procedures and functions) are always allocated on the stack and accessed relative to EBP, and the value of a local variable symbol is its signed offset from EBP. The assembler automatically adds [EBP] in references to local variables. For example, given the declaration

```
var Count: Integer;
```

within a function or procedure, the instruction

```
MOV     EAX,Count
```

assembles into MOV EAX,[EBP–4].

The built-in assembler treats **var** parameters as a 32-bit pointers, and the size of a **var** parameter is always 4. The syntax for accessing a **var** parameter is different from that for accessing a value parameter. To access the contents of a **var** parameter, you must first load the 32-bit pointer and then access the location it points to. For example,

```
function Sum(var X, Y: Integer): Integer; stdcall;
begin
  asm
    MOV     EAX,X
    MOV     EAX,[EAX]
    MOV     EDX,Y
    ADD     EAX,[EDX]
    MOV     @Result,EAX
  end;
end;
```

Identifiers can be qualified within **asm** statements. For example, given the declarations

```
type
  TPoint = record
    X, Y: Integer;
  end;
  TRect = record
    A, B: TPoint;
  end;
var
  P: TPoint;
  R: TRect;
```

the following constructions can be used in an **asm** statement to access fields.

```
MOV     EAX,P.X
MOV     EDX,P.Y
MOV     ECX,R.A.X
MOV     EBX,R.B.Y
```

A type identifier can be used to construct variables on the fly. Each of the following instructions generates the same machine code, which loads the contents of [EDX] into EAX.

```
MOV     EAX,(TRect PTR [EDX]).B.X
MOV     EAX,TRect([EDX]).B.X
MOV     EAX,TRect[EDX].B.X
MOV     EAX,[EDX].TRect.B.X
```

Expression classes

The built-in assembler divides expressions into three classes: *registers*, *memory references*, and *immediate values*.

An expression that consists solely of a register name is a *register* expression. Examples of register expressions are AX, CL, DI, and ES. Used as operands, register expressions direct the assembler to generate instructions that operate on the CPU registers.

Expressions that denote memory locations are memory references. Object Pascal's labels, variables, typed constants, procedures, and functions belong to this category.

Expressions that aren't registers and aren't associated with memory locations are immediate values. This group includes Object Pascal's untyped constants and type identifiers.

Immediate values and memory references cause different code to be generated when used as operands. For example,

```
const
  Start = 10;
var
  Count: Integer;
  ⋮
asm
  MOV     EAX,Start           { MOV EAX,xxxx }
  MOV     EBX,Count           { MOV EBX,[xxxx] }
  MOV     ECX,[Start]         { MOV ECX,[xxxx] }
  MOV     EDX,OFFSET Count    { MOV EDX,xxxx }
end;
```

Because *Start* is an immediate value, the first MOV is assembled into a move immediate instruction. The second MOV, however, is translated into a move memory instruction, as *Count* is a memory reference. In the third MOV, the brackets convert *Start* into a memory reference (in this case, the word at offset 10 in the data segment). In the fourth MOV, the OFFSET operator converts *Count* into an immediate value (the offset of *Count* in the data segment).

The brackets and OFFSET operator complement each other. The following **asm** statement produces identical machine code to the first two lines of the previous **asm** statement.

```
asm
  MOV     EAX,OFFSET [Start]
  MOV     EBX,[OFFSET Count]
end;
```

Memory references and immediate values are further classified as either *relocatable* or *absolute*. Relocation is the process by which the linker assigns absolute addresses to symbols. A relocatable expression denotes a value that requires relocation at link time, while an absolute expression denotes a value that requires no such relocation. Typically, expressions that refer to labels, variables, procedures, or functions are relocatable, since the final address of these symbols is unknown at compile time. Expressions that operate solely on constants are absolute.

The built-in assembler allows you to carry out any operation on an absolute value, but it restricts operations on relocatable values to addition and subtraction of constants.

Expression types

Every built-in assembler expression has a type—or, more correctly, a size, because the assembler regards the type of an expression simply as the size of its memory location. For example, the type of an *Integer* variable is four, because it occupies 4 bytes. The built-in assembler performs type checking whenever possible, so in the instructions

```
var
  QuitFlag: Boolean;
  OutBufPtr: Word;
  ⋮
asm
  MOV     AL,QuitFlag
  MOV     BX,OutBufPtr
end;
```

the assembler checks that the size of *QuitFlag* is one (a byte), and that the size of *OutBufPtr* is two (a word). The instruction

```
MOV     DL,OutBufPtr
```

produces an error because DL is a byte-sized register and *OutBufPtr* is a word. The type of a memory reference can be changed through a typecast; these are correct ways of writing the previous instruction:

```
MOV     DL,BYTE PTR OutBufPtr
MOV     DL,Byte(OutBufPtr)
MOV     DL,OutBufPtr.Byte
```

These MOV instructions all refer to the first (least significant) byte of the *OutBufPtr* variable.

In some cases, a memory reference is untyped. One example is an immediate value (Buffer) enclosed in square brackets:

```
procedure Example(var Buffer);
asm
  MOV AL, [Buffer]
  MOV CX, [Buffer]
  MOV EDX, [Buffer]
```

The built-in assembler permits these instructions, because the expression [Buffer] has no type—it just means “the contents of the location indicated by Buffer,” and the type can be determined from the first operand (byte for AL, word for CX, and double-word for EDX).

In cases where the type can’t be determined from another operand, the built-in assembler requires an explicit typecast. For example,

```
INC     BYTE PTR [ECX]
IMUL    WORD PTR [EDX]
```

The following table summarizes the predefined type symbols that the built-in assembler provides in addition to any currently declared Object Pascal types.

Table 13.5 Predefined type symbols

Symbol	Type
BYTE	1
WORD	2
DWORD	4

Table 13.5 Predefined type symbols (continued)

Symbol	Type
QWORD	8
TBYTE	10

Expression operators

The built-in assembler provides a variety of operators. Precedence rules are different from Object Pascal; for example, in an **asm** statement, AND has lower precedence than the addition and subtraction operators. The following table lists the built-in assembler's expression operators in decreasing order of precedence.

Table 13.6 Precedence of built-in assembler expression operators

Operators	Remarks	Precedence
&		highest
(), **[]**, **.**, **HIGH**, **LOW**		
+, **–**	unary + and –	
:		
OFFSET, **TYPE**, **PTR**, *****, **/**, **MOD**, **SHL**, **SHR**, **+**, **–**	binary + and –	
NOT, **AND**, **OR**, **XOR**		lowest

The following table defines the built-in assembler's expression operators.

Table 13.7 Definitions of built-in assembler expression operators

Operator	Description
&	**Identifier override**. The identifier immediately following the ampersand is treated as a user-defined symbol, even if the spelling is the same as a built-in assembler reserved symbol.
(...)	**Subexpression**. Expressions within parentheses are evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the parentheses; the result in this case is the sum of the values of the two expressions, with the type of the first expression.
[...]	**Memory reference**. The expression within brackets is evaluated completely prior to being treated as a single expression element. Another expression can precede the expression within the brackets; the result in this case is the sum of the values of the two expressions, with the type of the first expression. The result is always a memory reference.
.	**Structure member selector**. The result is the sum of the expression before the period and the expression after the period, with the type of the expression after the period. Symbols belonging to the scope identified by the expression before the period can be accessed in the expression after the period.
HIGH	Returns the high-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.
LOW	Returns the low-order 8 bits of the word-sized expression following the operator. The expression must be an absolute immediate value.

Table 13.7 Definitions of built-in assembler expression operators (continued)

Operator	Description
+	**Unary plus**. Returns the expression following the plus with no changes. The expression must be an absolute immediate value.
–	**Unary minus**. Returns the negated value of the expression following the minus. The expression must be an absolute immediate value.
+	**Addition**. The expressions can be immediate values or memory references, but only one of the expressions can be a relocatable value. If one of the expressions is a relocatable value, the result is also a relocatable value. If either of the expressions is a memory reference, the result is also a memory reference.
–	**Subtraction**. The first expression can have any class, but the second expression must be an absolute immediate value. The result has the same class as the first expression.
:	**Segment override**. Instructs the assembler that the expression after the colon belongs to the segment given by the segment register name (CS, DS, SS, FS, GS, or ES) before the colon. The result is a memory reference with the value of the expression after the colon. When a segment override is used in an instruction operand, the instruction is prefixed with an appropriate segment-override prefix instruction to ensure that the indicated segment is selected.
OFFSET	Returns the offset part (double word) of the expression following the operator. The result is an immediate value.
TYPE	Returns the type (size in bytes) of the expression following the operator. The type of an immediate value is 0.
PTR	**Typecast operator**. The result is a memory reference with the value of the expression following the operator and the type of the expression in front of the operator.
*	**Multiplication**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
/	**Integer division**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
MOD	**Remainder after integer division**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHL	**Logical shift left**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
SHR	**Logical shift right**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
NOT	**Bitwise negation.** The expression must be an absolute immediate value, and the result is an absolute immediate value.
AND	**Bitwise AND**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
OR	**Bitwise OR**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.
XOR	**Bitwise exclusive OR**. Both expressions must be absolute immediate values, and the result is an absolute immediate value.

Assembly procedures and functions

You can write complete procedures and functions using inline assembly language code, without including a **begin...end** statement. For example,

```
function LongMul(X, Y: Integer): Longint;
asm
  MOV     EAX,X
  IMUL    Y
end;
```

The compiler performs several optimizations on these routines:

- No code is generated to copy value parameters into local variables. This affects all string-type value parameters and other value parameters whose size isn't 1, 2, or 4 bytes. Within the routine, such parameters must be treated as if they were **var** parameters.
- Unless a function returns a string, variant, or interface reference, the compiler doesn't allocate a function result variable; a reference to the *@Result* symbol is an error. For strings, variants, and interfaces, the caller always allocates an *@Result* pointer.
- The compiler only generates stack frames for nested routines, for routines that have local parameters, or for routines that have parameters on the stack.
- The automatically generated entry and exit code for the routine looks like this:

  ```
  PUSH    EBP             ;Present if Locals <> 0 or Params <> 0
  MOV     EBP,ESP         ;Present if Locals <> 0 or Params <> 0
  SUB     ESP,Locals      ;Present if Locals <> 0
   ⋮
  MOV     ESP,EBP         ;Present if Locals <> 0
  POP     EBP             ;Present if Locals <> 0 or Params <> 0
  RET     Params          ;Always present
  ```

  If locals include variants, long strings, or interfaces, they are initialized to zero but not finalized.
- *Locals* is the size of the local variables and *Params* is the size of the parameters. If both *Locals* and *Params* are zero, there is no entry code, and the exit code consists simply of a RET instruction.

Assembly language functions return their results as follows.

- Ordinal values are returned in AL (8-bit values), AX (16-bit values), or EAX (32-bit values).
- Real values are returned in ST(0) on the coprocessor's register stack. (*Currency* values are scaled by 10000.)
- Pointers, including long strings, are returned in EAX.
- Short strings and variants are returned in the temporary location pointed to by *@Result*.

Appendix A

Object Pascal grammar

```
Goal -> (Program | Package | Library | Unit)

Program -> [PROGRAM Ident ['(' IdentList ')'] ';']
           ProgramBlock '.'

Unit -> UNIT Ident [PortabilityDirective] ';'
        InterfaceSection
        ImplementationSection
        InitSection '.'

Package -> PACKAGE Ident ';'
           [RequiresClause]
           [ContainsClause]
           END '.'

Library -> LIBRARY Ident ';'
           ProgramBlock '.'

ProgramBlock -> [UsesClause]
                Block

UsesClause -> USES IdentList ';'

PortabilityDirective -> platform
                     -> deprecated
                     -> library

InterfaceSection -> INTERFACE
                    [UsesClause]
                    [InterfaceDecl]...

InterfaceDecl -> ConstSection
              -> TypeSection
              -> VarSection
              -> ExportedHeading

ExportedHeading -> ProcedureHeading ';' [Directive]
                -> FunctionHeading ';' [Directive]
```

```
ImplementationSection -> IMPLEMENTATION
                         [UsesClause]
                         [DeclSection]...
                         [ExportsStmt]...

Block -> [DeclSection]
         [ExportsStmt]...
         CompoundStmt
         [ExportsStmt]...

ExportsStmt -> EXPORTS ExportsItem [, ExportsItem]...

ExportsItem -> Ident [NAME|INDEX “'” ConstExpr “'”]
                     [INDEX|NAME “'” ConstExpr “'”]

DeclSection -> LabelDeclSection
            -> ConstSection
            -> TypeSection
            -> VarSection
            -> ProcedureDeclSection

LabelDeclSection -> LABEL LabelId

ConstSection -> CONST (ConstantDecl ';')...

ConstantDecl -> Ident '=' ConstExpr [PortabilityDirective]
             -> Ident ':' TypeId '=' TypedConstant [PortabilityDirective]

TypeSection -> TYPE (TypeDecl ';')

TypeDecl -> Ident '=' [TYPE] Type [PortabilityDirective]
         -> Ident '=' [TYPE] RestrictedType [PortabilityDirective]

TypedConstant -> (ConstExpr | ArrayConstant | RecordConstant)

ArrayConstant -> '(' TypedConstant ',' ')'

RecordConstant -> '(' RecordFieldConstant ';'... ')'

RecordFieldConstant -> Ident ':' TypedConstant

Type -> TypeId
     -> SimpleType
     -> StrucType
     -> PointerType
     -> StringType
     -> ProcedureType
     -> VariantType
     -> ClassRefType

RestrictedType -> ObjectType
               -> ClassType
               -> InterfaceType

ClassRefType -> CLASS OF TypeId

SimpleType -> (OrdinalType | RealType)
```

```
RealType -> REAL48
         -> REAL
         -> SINGLE
         -> DOUBLE
         -> EXTENDED
         -> CURRENCY
         -> COMP

OrdinalType -> (SubrangeType | EnumeratedType | OrdIdent)

OrdIdent -> SHORTINT
         -> SMALLINT
         -> INTEGER
         -> BYTE
         -> LONGINT
         -> INT64
         -> WORD
         -> BOOLEAN
         -> CHAR
         -> WIDECHAR
         -> LONGWORD
         -> PCHAR

VariantType -> VARIANT
            -> OLEVARIANT

SubrangeType -> ConstExpr '..' ConstExpr

EnumeratedType -> '(' EnumeratedTypeElement ','... ')'

EnumeratedTypeElement -> Ident [ '=' ConstExpr ]

StringType -> STRING
           -> ANSISTRING
           -> WIDESTRING
           -> STRING '[' ConstExpr ']'

StrucType -> [PACKED] (ArrayType [PACKED]| SetType | FileType | RecType [PACKED])

ArrayType -> ARRAY ['[' OrdinalType ','... ']'] OF Type [PortabilityDirective]

RecType -> RECORD [FieldList] END [PortabilityDirective]

FieldList -> FieldDecl ';'... [VariantSection] [';']

FieldDecl -> IdentList ':' Type [PortabilityDirective]

VariantSection -> CASE [Ident ':'] TypeId OF RecVariant ';'...

RecVariant -> ConstExpr ','... ':' '(' [FieldList] ')'

SetType -> SET OF OrdinalType [PortabilityDirective]

FileType -> FILE OF TypeId [PortabilityDirective]

PointerType -> '^' TypeId [PortabilityDirective]

ProcedureType -> (ProcedureHeading | FunctionHeading) [OF OBJECT]
```

```
VarSection -> VAR (VarDecl ';')...

VarDecl
  On Windows -> IdentList ':' Type [(ABSOLUTE (Ident | ConstExpr)) | '=' ConstExpr]
    [PortabilityDirective]
  On Linux   -> IdentList ':' Type [ABSOLUTE (Ident) | '=' ConstExpr] [PortabilityDirective]

Expression -> SimpleExpression [RelOp SimpleExpression]...

SimpleExpression -> ['+' | '-'] Term [AddOp Term]...

Term -> Factor [MulOp Factor]...

Factor -> Designator ['(' ExprList ')']
       -> '@' Designator
       -> Number
       -> String
       -> NIL
       -> '(' Expression ')'
       -> NOT Factor
       -> SetConstructor
       -> TypeId '(' Expression ')'

RelOp -> '>'
      -> '<'
      -> '<='
      -> '>='
      -> '<>'
      -> IN
      -> IS
      -> AS

AddOp -> '+'
      -> '-'
      -> OR
      -> XOR

MulOp -> '*'
      -> '/'
      -> DIV
      -> MOD
      -> AND
      -> SHL
      -> SHR

Designator -> QualId ['.' Ident | '[' ExprList ']' | '^']...

SetConstructor -> '[' [SetElement ','...] ']'

SetElement -> Expression ['..' Expression]

ExprList -> Expression ','...

Statement -> [LabelId ':'] [SimpleStatement | StructStmt]

StmtList -> Statement ';'

SimpleStatement -> Designator ['(' [ExprList] ')']
                -> Designator ':=' Expression
                -> INHERITED
                -> GOTO LabelId
```

```
StructStmt -> CompoundStmt
           -> ConditionalStmt
           -> LoopStmt
           -> WithStmt
           -> TryExceptStmt
           -> TryFinallyStmt
           -> RaiseStmt
           -> AssemblerStmt

CompoundStmt -> BEGIN StmtList END

ConditionalStmt -> IfStmt
                -> CaseStmt

IfStmt -> IF Expression THEN Statement [ELSE Statement]

CaseStmt -> CASE Expression OF CaseSelector ';'... [ELSE StmtList] [';'] END

CaseSelector -> CaseLabel ','... ':' Statement

CaseLabel -> ConstExpr ['..' ConstExpr]

LoopStmt -> RepeatStmt
         -> WhileStmt
         -> ForStmt

RepeatStmt -> REPEAT Statement UNTIL Expression

WhileStmt -> WHILE Expression DO Statement

ForStmt -> FOR QualId ':=' Expression (TO | DOWNTO) Expression DO Statement

WithStmt -> WITH IdentList DO Statement

TryExceptStmt -> TRY
                   Statement...
                 EXCEPT
                   ExceptionBlock
                 END

ExceptionBlock -> [ON [Ident ‘:’] TypeID DO Statement]...
                  [ELSE Statement...]

TryFinallyStmt -> TRY
                   Statement
                  FINALLY
                   Statement
                  END

RaiseStmt -> RAISE [object] [AT address]

AssemblerStatement -> ASM
                   -> <assemblylanguage>
                   -> END

ProcedureDeclSection -> ProcedureDecl
                     -> FunctionDecl

ProcedureDecl -> ProcedureHeading ';' [Directive] [PortabilityDirective]
                 Block ';'

FunctionDecl -> FunctionHeading ';' [Directive] [PortabilityDirective]
                Block ';'
```

```
FunctionHeading -> FUNCTION Ident [FormalParameters] ':' (SimpleType | STRING)

ProcedureHeading -> PROCEDURE Ident [FormalParameters]

FormalParameters -> '(' [FormalParm ';'...] ')'

FormalParm -> [VAR | CONST | OUT] Parameter

Parameter -> IdentList [':' ([ARRAY OF] SimpleType | STRING | FILE)]
          -> Ident ':' SimpleType '=' ConstExpr

Directive -> CDECL
          -> REGISTER
          -> DYNAMIC
          -> VIRTUAL
          -> EXPORT
          -> EXTERNAL
          -> NEAR
          -> FAR
          -> FORWARD
          -> MESSAGE ConstExpr
          -> OVERRIDE
          -> OVERLOAD
          -> PASCAL
          -> REINTRODUCE
          -> SAFECALL
          -> STDCALL
          -> VARARGS
          -> LOCAL
          -> ABSTRACT

ObjectType -> OBJECT [ObjHeritage] [ObjFieldList] [MethodList] END

ObjHeritage -> '(' QualId ')'

MethodList -> (MethodHeading [';' VIRTUAL]) ';'...

MethodHeading -> ProcedureHeading
              -> FunctionHeading
              -> ConstructorHeading
              -> DestructorHeading

ConstructorHeading -> CONSTRUCTOR Ident [FormalParameters]

DestructorHeading -> DESTRUCTOR Ident [FormalParameters]

ObjFieldList -> (IdentList ':' Type) ';'

InitSection -> INITIALIZATION StmtList [FINALIZATION StmtList] END
            -> BEGIN StmtList END
            -> END

ClassType -> CLASS [ClassHeritage]
             [ClassVisibility]
             [ClassFieldList]
             [ClassMethodList]
             [ClassPropertyList]
             END

ClassHeritage -> '(' IdentList ')'

ClassVisibility -> [PUBLIC | PROTECTED | PRIVATE | PUBLISHED]
```

```
ClassFieldList -> (ClassVisibility ObjFieldList) ';'...

ClassMethodList -> (ClassVisibility MethodList) ';'...

ClassPropertyList -> (ClassVisibility PropertyList ';')...

PropertyList -> PROPERTY Ident [PropertyInterface] [PropertySpecifiers]
   [PortabilityDirective]

PropertyInterface -> [PropertyParameterList] ':' Ident

PropertyParameterList -> '[' (IdentList ':' TypeId) ';'... ']'

PropertySpecifiers -> [INDEX ConstExpr]
                      [READ Ident]
                      [WRITE Ident]
                      [STORED (Ident | Constant)]
                      [(DEFAULT ConstExpr) | NODEFAULT]
                      [IMPLEMENTS TypeId]

InterfaceType -> INTERFACE [InterfaceHeritage]
                 [ClassMethodList]
                 [ClassPropertyList]
                 ...
                 END

InterfaceHeritage -> '(' IdentList ')'

RequiresClause -> REQUIRES IdentList... ';'

ContainsClause -> CONTAINS IdentList... ';'

IdentList -> Ident ','...

QualId -> [UnitId '.'] Ident

TypeId -> [UnitId '.'] <type-identifier>

Ident -> <identifier>

ConstExpr -> <constant-expression>

UnitId -> <unit-identifier>

LabelId -> <label-identifier>

Number -> <number>

String -> <string>
```

Index

Symbols

- 4-4, 4-6, 4-9, 4-10
" 13-9
4-4
$ 4-4, 4-5
(*, *) 4-5
(,) 4-2, 4-13, 4-14, 5-6, 5-43, 6-2, 6-3, 6-11, 7-2, 10-1
* 4-2, 4-6, 4-10
+ 4-4, 4-6, 4-9, 4-10
, 3-6, 4-24, 5-6, 5-21, 5-23, 6-11, 7-17, 9-5, 9-9, 10-7, 13-2
. 3-2, 4-2, 4-14, 5-27, 9-9, 10-5
/ 4-2, 4-6
// 4-5
: 4-2, 4-17, 4-24, 5-21, 5-23, 5-38, 5-43, 6-3, 6-11, 7-17, 7-19, 7-29, 13-2
:= 4-17, 4-26
  named parameters 10-12
; 3-2, 3-6, 4-16, 4-17, 4-20, 4-25, 5-21, 5-23, 5-38, 6-2, 6-3, 6-11, 7-2, 7-6, 9-5, 9-9, 10-1, 10-4, 10-10
  before ’else’ 4-23
< 4-2, 4-9, 4-11
<= 4-9, 4-10, 4-11
<> 4-9, 4-10, 4-11
= 4-2, 4-9, 4-10, 4-11, 4-17, 5-37, 5-38, 5-40, 5-43, 6-11, 6-18, 10-5
> 4-2, 4-9, 4-11
>= 4-9, 4-10, 4-11
@ 4-6, 4-12, 5-26, 5-30, 5-44, 7-17
@@ 5-30
@Result 13-10, 13-17
[,] 4-14, 5-11, 5-12, 5-17, 5-31, 6-15, 6-20, 7-17, 7-19, 7-20, 10-1, 10-10
^ 4-6, 4-9, 5-19, 5-27
  and variants 5-33
  pointer overview 5-26
_ 4-2
{, } 4-5, 10-1, 10-10
’ 4-4, 9-9, 10-1, 10-10

A

$A directive 11-8
absolute (directive) 5-39
absolute addresses 5-39
absolute expressions (assembler) 13-13
abstract methods 7-12
access specifiers 7-1, 7-17
  array properties 7-19
  Automation 7-6
  calling convention 6-5, 7-18
  index specifiers and 7-21
  overloading 7-13, 7-18
  overriding 7-22
actual parameters 6-19
Add method (TCollection) 7-9
addition 4-6
  pointers 4-9
Addr function 5-27
_AddRef method 10-2, 10-5, 10-9
address operator 4-12, 5-26, 5-30, 5-44
  properties and 7-17
alignment (data) 5-17, 11-8
  See also internal data formats
AllocMemCount variable 11-2
AllocMemSize variable 11-2
alphanumeric characters 4-1, 4-2
ampersand *See* Symbols
ancestors 7-3
and 4-7, 4-8
anonymous values (enumerated types) 5-8, 7-5
ANSI characters 5-5, 5-12, 5-13
AnsiChar type 5-5, 5-11, 5-13, 5-27, 11-3
AnsiString type 5-10, 5-12, 5-13, 5-15, 5-28
  See also long strings
  memory management 11-6
  variant arrays and 5-34
Append procedure 8-2, 8-4, 8-5, 8-6
application partitioning 9-8
Application variable 2-5, 3-3
arithmetic operators 4-6, 5-4
array properties 7-5, 7-19
  default 7-20
  in dispatch interfaces 10-11
  storage specifiers and 7-22
arrays 5-3, 5-18 to 5-21
  ’array of const’ 6-17
  accessing with PByteArray 5-28
  accessing with PWordArray 5-28
  assignments and 5-21
  character 4-5, 5-13, 5-14, 5-17, 5-18
  character arrays and string constants 4-5, 5-14, 5-43
  constant 5-43
  dynamic 5-19, 5-40, 6-16, 11-7
  indexes 4-14
  multidimensional 5-18, 5-20
  open array constructors 6-17, 6-20
  parameters 6-12, 6-15
  static 5-18, 11-7
  variants and 5-31, 5-34
as 4-12, 7-24, 7-25, 10-10
ASCII 4-1, 4-4, 4-5, 5-13
asm statements 13-1, 13-16
assembler (directive) 6-6, 13-1
assembly language
  assembler routines 13-16
  built-in assembler 13-1 to 13-17
  external routines 6-7
  Object Pascal and 13-5, 13-7, 13-8, 13-10, 13-13, 13-15
Assert procedure 7-27
assertions 7-27
Assign procedure
  custom 8-4
Assigned function 5-31, 10-9
AssignFile procedure 8-2, 8-4, 8-6
assignment statements 4-17
  typecasts 4-15
assignment-compatibility 5-36, 7-3, 10-9
asterisk *See* Symbols
at (reserved word) 7-27
at-sign *See* @, address operator
automatable types 7-6, 10-11
automated class members 7-4, 7-6
Automation 7-6, 10-10 to 10-13
  See also COM
  dual interfaces 10-13
  method calls 10-12
  variants and 11-12

B

$B directive 4-8
base types 5-8, 5-17, 5-18, 5-19
begin (reserved word) 3-2, 4-20, 6-2, 6-3
binary operators 4-6
binding
  fields 7-7
  methods 7-10
bitwise operators,not 4-8
blanks 4-1
BlockRead procedure 8-4
blocks 4-27 to 4-28
  function 3-4, 6-1, 6-3
  library 9-5
  outer and inner 4-29
  procedure 3-4, 6-1, 6-2
  program 3-1, 3-2
  scope 4-27 to 4-29
  try...except 7-28, 7-30
  try...finally 7-31
BlockWrite procedure 8-4
body (routine) 6-1
boldface 1-2
Boolean operators 4-7
  complete vs. partial evaluation 4-7
Boolean type 5-6, 11-3
Boolean types 5-5, 11-3
BORLANDMM.DLL 9-8
braces *See* Symbols
brackets *See* Symbols
Break procedure 4-25
  exception handlers 7-29
  in try...finally block 7-32
BSTR type (COM) 5-13
built-in assembler 13-1 to 13-17
built-in types 5-1
by reference (parameters) 6-12, 6-13, 10-12, 12-1
by value (parameters) 6-12, 10-12, 12-1
Byte type 5-4, 11-3
  assembler 13-14
ByteBool type 5-6, 11-3

C

C++ 6-6, 10-1, 11-11
calling conventions 5-29, 6-4, 12-1
  access specifiers 6-5, 7-18
  interfaces 10-3, 10-7
  methods 12-3
  shared libraries 9-4
  varargs directive 6-7
calling routines 9-1
Cardinal type 5-3
caret *See* Symbols
carriage-return 4-1, 4-5
case (reserved word) 4-24, 5-23
case statements 4-24
case-sensitivity 4-1, 4-2, 6-8
  unit names and files 4-2
-cc compiler switch 8-3
cdecl (calling convention) 6-4, 12-2
  constructors and destructors 12-4
  Self 12-4
  varargs 6-7
Char type 5-5, 5-13, 5-27, 11-3
character operators 4-9
character sets
  ANSI 5-5, 5-12, 5-13
  extended 5-13
  multibyte (MBCS) 5-13
  Pascal 4-1, 4-2, 4-4
  single-byte (SBCS) 5-13
character strings 4-1, 4-4, 5-45
characters
  pointers 5-27
  string literals as 4-5, 5-5
  types 5-5, 11-3
  wide 5-13, 11-3
checked typecasts
  interfaces 10-10
  objects 7-25
Chr function 5-5
circular references
  packages 9-10
  units 3-7 to 3-9
circumflex *See* Symbols
classes 7-1 to 7-32
  class methods 7-1, 7-25
  class references 7-23
  class types 7-1, 7-2
  comparison 4-11
  compatibility 7-3, 10-9
  declaring class types 7-2, 7-4, 7-6, 7-7, 7-8, 7-17, 10-5
  files and 5-25
  memory 11-10
  metaclasses 7-23
  operators 4-12, 7-24
  scope 4-29
  variants and 5-31
Classes unit 7-9, 7-23
ClassParent method 7-24
class-reference types 7-23
  comparison 4-11
  constructors and 7-24
  memory 11-11
  variants and 5-31
ClassType method 7-24
clients 3-4
Close function 8-4, 8-6
CloseFile function 8-5
CloseFile procedure 8-6
CLX 1-2
CmdLine variable 9-7
colon *See* Symbols
COM 10-4
  See also Automation
  interfaces 10-2, 10-10 to 10-13
  out parameters 6-14
  variants and 5-31, 5-33, 11-11
COM error handling 6-5
comma *See* Symbols
comments 4-1, 4-5
ComObj unit 7-6, 10-11
Comp type 5-9, 5-10, 11-5
comparison
  classes 4-11
  class-reference types 4-11
  dynamic arrays 5-20
  integer types 4-11
  objects 4-11
  packed strings 4-11
  PChar type 4-11
  real types 4-11
  relational operators 4-11
  strings 4-11, 5-11
compiler 2-2, 2-3, 2-5, 3-1
  command-line 2-3 to 2-5
  directives 3-2, 4-5
  packages 9-11
compile-time binding *See* static methods
complete evaluation 4-7
components, of classes *See* members
compound statements 4-20
concatenation (strings) 4-9
conditional statements 4-20
conjunction 4-7
console applications 2-3, 8-3
const (reserved word) 5-40, 5-43, 6-11, 6-12, 6-13, 6-17, 12-1
constant expression
  array constants 5-43
  case statements 4-24
  constant declarations 5-40, 5-43

- default paramters 6-18
- defined 5-41
- enumerated types 5-7
- initializing variables 5-38
- subrange types 5-8, 5-9
- type 5-41, 6-8

constant parameters 6-12, 6-13, 6-19, 12-1
- open array constructors 6-20

constants 4-6, 5-40
- array 5-43
- assembler 13-9
- declared 5-40 to 5-45
- numeric *See* numerals
- pointer 5-44
- procedural 5-44
- record 5-43
- true 5-40
- type compatibility 5-40
- typed 5-42

constructors 7-1, 7-9, 7-13
- calling conventions 12-4
- class references and 7-24
- exceptions 7-28, 7-32

contains clause 9-9, 9-10
context-sensitive Help (error-handling) 7-32
Continue procedure 4-25
- exception handlers 7-29
- in try...finally block 7-32

control (program) 6-19, 12-1 to 12-6
control characters 4-1, 4-5
control loops 4-20, 4-25
control strings 4-4
conversion
- *See also* typecasts
- variants 5-31, 5-32 to 5-33, 5-34

Copy function 5-20
copy-on-write semantics 5-12
CORBA
- interfaces 10-3
- out parameters 6-14

CPU *See* registers
Create method 7-13
Currency type 5-9, 5-10, 5-28, 11-5

D

data alignment 5-17, 11-8
data formats, internal 11-2 to 11-12
data types *See* types
.dcp files 9-11, 2-3
.dcu files 2-3, 3-7, 9-10, 9-11
Dec procedure 5-3, 5-4
declarations 4-1, 4-16, 4-28
- class 7-2, 7-7, 7-8, 7-17, 10-5
- constant 5-40, 5-43
- defining 6-6, 7-6, 7-8, 10-4
- field 7-7
- forward 3-4, 6-6, 7-6, 10-4
- function 6-1, 6-3
- implementation 7-8
- interface 3-4
- local 6-10
- method 7-8
- package 9-9
- procedure 6-1, 6-2
- property 7-17, 7-19
- type 5-37
- variable 5-38

declared types 5-1
decrementing ordinals 5-3, 5-4, 5-5
default (directive) 7-20, 10-11
default parameters 6-11, 6-18 to 6-19
- Automation objects 10-12
- forward and interface declarations 6-19
- overloading and 6-10, 6-19
- procedural types 6-18

default properties 7-20
- interfaces 10-2

default property (COM object) 5-33
default specifier 7-6, 7-17, 7-21
DefaultHandler method 7-16, 7-17
defining declarations 6-6, 7-6, 7-8, 10-4
DefWindowProc function 7-16
delegated interface 10-7
delegation (interface implementation) 10-6
$DENYPACKAGEUNIT directive 9-12
dependency
- units 3-7 to 3-9

deprecated (directive) 4-17
dereference operator 4-9, 5-19
- pointer overview 5-26
- variants and 5-33

descendants 7-3, 7-5
$DESIGNONLY directive 9-12
design-time packages 9-8
.desk files 2-2
desktop settings files 2-2
Destroy method 7-13, 7-15, 7-29
destructors 7-1, 7-13, 7-14
- calling conventions 12-4

device drivers, text-file 8-4
device functions 8-4, 8-5
.dfm files 2-2, 2-7, 7-5
difference (sets) 4-10
directives 4-1, 4-3
- *See also* reserved words
- assembler 13-3
- compiler 3-2, 4-5
- list 4-3
- order 7-8

directory paths
- in uses clause 3-6

disjunction 4-7
- bitwise 4-8

dispatch interface types 10-10
Dispatch method 7-16
dispatching messages 7-16
dispatching method calls 7-11
dispid (directive) 7-6, 10-2, 10-11
dispinterface 10-10
dispinterface (reserved word) 10-2
Dispose procedure 5-19, 5-39, 7-4, 9-8, 11-1, 11-2
div 4-6
division 4-6
dlclose 9-2
.DLL files 6-7, 9-1
DLL_PROCESS_DETACH 9-7
DLL_THREAD_ATTACH 9-7
DLL_THREAD_DETACH 9-7
DLLProc variable 9-7
DLLs 9-1 to 9-8
- calling routines in 6-7
- dynamic arrays in 9-8
- dynamic variables in 9-8
- exceptions 9-7
- global variables 9-6
- loading dynamically 9-2
- loading statically 9-1
- long strings in 9-8
- multithreading 9-7
- variables 9-1
- writing 9-3

dlopen 9-2
dlsym 9-2
DMTINDEX 13-6
do (reserved word) 4-21, 4-26, 7-29
.dof files 2-2
dollar *See* Symbols

Double type 5-9, 11-5
downto (reserved word) 4-26
.dpk files 2-2, 9-11
.dpr files 2-2, 3-1, 3-6
.dpu files 2-3, 3-7, 9-10, 9-11
.drc files 2-3
.dsk files 2-2
dual interfaces 10-3, 10-13
  methods 6-5
DWORD type (assembler) 13-14
dynamic arrays 5-19, 11-7
  assigning to 5-19
  comparison 5-20
  files and 5-25
  freeing 5-40
  in dynamically loadable libraries 9-8
  memory management 11-2
  multidimensional 5-20
  open array parameters and 6-16
  records and 5-23
  truncating 5-20
  variants and 5-31
dynamic methods 7-10
dynamic variables 5-39
  and pointer constants 5-45
  in dynamically loadable libraries 9-8
dynamically loadable libraries 6-7, 9-1 to 9-12
  dynamic arrays 9-8
  dynamic variables 9-8
  exceptions 9-7
  global variables 9-6
  loading statically 9-1
  long strings 9-8
  variables 9-1
  writing 9-3
dynamic-link libraries *See* DLLs

E

E (in numerals) 4-4
EAssertionFailed 7-27
else (reserved word) 4-22, 4-24, 7-28
empty set 5-17
end (reserved word) 3-2, 4-20, 4-24, 5-21, 5-23, 6-2, 6-3, 7-2, 7-28, 7-32, 9-9, 10-1, 10-10, 13-1
end-of-line character 4-1, 8-3
enumerated types 5-6 to 5-8, 11-3
  anonymous values 5-8, 7-5
  publishing 7-5
Eof function 8-5
Eoln function 8-5
equality operator 4-11
error handling *See* exceptions
ErrorAddr variable 12-5
EStackOverflow exception 11-2
EVariantError exception 5-33
event handlers 2-7, 7-5
events 2-7, 7-5
example programs 2-3 to 2-5
except (reserved word) 7-28
ExceptAddr function 7-32
Exception class 7-27, 7-32
exception firewalls 6-5
exception handlers 7-26, 7-29
  identifiers in 7-30
ExceptionInformation variable 9-8
exceptions 4-20, 7-13, 7-15, 7-26 to 7-32
  constructors 7-28, 7-32
  declaring 7-27
  destroying 7-28, 7-29
  dynamically loadable libraries 9-6, 9-7
  file I/O 8-3
  handling 7-28, 7-30, 7-32
  in initialization section 7-28
  nested 7-31
  propagation 7-29, 7-31, 7-32
  raising 7-27
  re-raising 7-30
  standard exceptions 7-32
  standard routines 7-32
ExceptObject function 7-32
executable files 2-3
Exit procedure 6-1
  exception handlers 7-29
  in try...finally block 7-32
exit procedures 9-6, 12-4 to 12-6
  packages and 12-4
ExitCode variable 9-6, 12-5
ExitProc variable 9-6, 12-4, 12-5
export (directive) 6-5
exports clause 4-28, 9-5
  overloaded routines 9-5
expressions 4-1, 4-5
  assembler 13-7 to 13-16
extended syntax 4-5, 4-18, 5-14, 6-1, 6-4
Extended type 4-7, 5-9, 5-10, 5-28, 11-5
external (directive) 6-6, 9-1, 9-2

F

False 5-6, 11-3
far (directive) 6-5
fields 5-21 to 5-24, 7-1, 7-7
  See also records, classes
  publishing 7-5
file (reserved word) 5-25
file I/O 8-1 to 8-6
  exceptions 8-3
file variables 8-2
FilePos function 8-2
files
  as parameters 6-12
  file types 5-25, 8-2
  generated 2-2, 2-3, 9-10, 9-11
  initializing 5-38
  memory 11-9
  source code 2-1
  text 8-2, 8-3
  typed 5-25, 8-2
  untyped 5-25, 8-2, 8-4
  variants and 5-31
FileSize function 8-2
finalization section 3-3, 3-5, 12-4
Finalize procedure 5-19
finally (reserved word) 7-32
floating-point types *See* real types
Flush function 8-4, 8-6
for statements 4-20, 4-25, 4-26
form files 2-2, 2-5, 3-1, 7-5, 7-21
formal parameters 6-19
forms 2-2
forward declarations
  classes 7-6
  default parameters 6-19
  interfaces 10-4
  overloading and 6-9
  routines 3-4, 6-6
Free method 7-15
FreeLibrary function 9-2
FreeMem procedure 5-39, 9-8, 11-1, 11-2
functions 3-4, 6-1 to 6-20
  assembler 13-16
  calling externally 6-6
  declaring 6-3, 6-6
  function calls 4-13, 4-18, 6-1, 6-19 to 6-20
  nested 5-29, 6-11
  overloading 6-6, 6-8
  pointers 4-12, 5-29
  return type 6-3, 6-4
  return value 6-3, 6-4

  return values in
    registers 12-3, 13-17
fundamental types 5-1

G

-$G- compiler switch 9-12
$G directive 9-12
generic types 5-1
GetHeapStatus function 11-2
GetMem procedure 5-27, 5-39, 9-8, 11-1, 11-2
GetMemoryManager procedure 11-2
GetProcAddress function 9-2
getter *See* read specifier
global identifiers 4-28
global variables 5-38
  dynamically loadable
    libraries 9-6
  interfaces 10-9
  memory management 11-2
GlobalAlloc 11-1
globally unique identifiers *See* GUIDs
goto statements 4-18
grammar (formal) A-1 to A-7
greater-than *See* Symbols
GUIDs 10-1, 10-3, 10-10
  generating 10-3

H

$H directive 5-11, 6-15
Halt procedure 12-4, 12-5
heading
  program 2-1, 3-1, 3-2
  routine 6-1
  unit 3-3
heap memory 5-39, 11-2
Hello world! 2-3
HelpContext property 7-32
hexadecimal numerals 4-4
hiding class members 7-8, 7-11, 7-22
  See also overloaded methods
  reintroduce 7-12
hiding interface implementations 10-6
High function 5-3, 5-4, 5-12, 5-18, 5-20, 6-16
HInstance variable 9-7
hint directives 4-17
$HINTS directive 4-17

I

$I directive 8-3
IDE
  See also Delphi
identifiers 4-1, 4-2, 4-3
  global and local 4-28
  in exception handlers 7-30
  qualified 3-7
  scope 4-27 to 4-29
IDispatch 10-9, 10-10
  dual interfaces 10-13
if...then statements 4-22
  nested 4-23
IInterface 10-2
immediate values (assembler) 13-12
implementation section 3-3, 3-4, 3-7
  and forward declarations 6-6
  methods 7-8
  scope 4-29
  uses clause 3-7
implements (directive) 7-22, 10-6
$IMPLICITBUILD directive 9-12
$IMPORTEDDATA directive 9-12
importing routines from libraries 9-1
in (reserved word) 3-6, 4-10, 5-17, 5-33, 9-9
Inc procedure 5-3, 5-4
incrementing ordinals 5-3, 5-4, 5-5
index (directive) 6-8
index specifier 7-6, 7-17, 7-20
index specifier (Windows only) 9-5
indexes 4-14
  array 5-18, 5-19, 5-20
  array properties 7-19
  in var parameters 5-34, 6-13
  string 5-11
  string variants 5-31
  variant arrays 5-34
indirect unit references 3-7
inequality operator 4-11
inheritance 7-2, 7-3, 7-5
  interfaces 10-2
inherited (reserved word) 7-9, 7-13
  calling conventions 12-4
  message handlers 7-16
InheritsFrom method 7-24
initialization
  dynamically loadable
    libraries 9-5
  files 5-38
  objects 7-13
  units 3-4
  variables 5-38
  variants 5-31, 5-38
initialization section 3-3, 3-4
  exceptions 7-28
Initialize procedure 5-39
inline (reserved word) 13-1
inline assembler code 13-1 to 13-17
inner block 4-29
InOut function 8-4, 8-5
input (program parameter) 3-2
input *See* file I/O
Input variable 8-3
Int64 type 4-7, 5-3, 5-4, 5-10, 11-3
  standard functions and
    procedures 5-4
  variants and 5-31
integer operators 4-6
Integer type 4-7, 5-3, 5-4
integer types 5-3
  comparison 4-11
  constants 5-41
  conversion 4-15
  data formats 11-3
integrated development environment *See* IDE
interface declarations 3-4
  default paramters 6-19
interface section 3-3, 3-4, 3-7
  forward declarations and 6-6
  methods 7-8
  scope 4-29
  uses clause 3-7
interfaces 7-2, 10-1 to 10-13
  accessing 10-8 to 10-10
  Automation 10-10
  calling conventions 10-3
  compatibility 10-9
  delegation 10-6
  dispatch interface
    types 10-10
  dual interfaces 10-13
  freeing 5-40
  GUIDs 10-1, 10-3, 10-10
  implementing 10-4 to 10-7
  interface references 10-8 to
    10-10

  interface types 10-1 to 10-4
  memory management 11-2
  method resolution clauses 10-5, 10-6
  properties 10-1, 10-4, 10-7
  querying 10-10
  records and 5-23
  typecasts 10-10
internal data formats 11-2 to 11-12
intersection (sets) 4-10
IntToHex function 5-4
IntToStr function 5-4
Invoke method 10-10
IOResult function 8-3, 8-4
is 4-12, 5-33, 7-24
IsLibrary variable 9-7
italics 1-2
IUnknown 10-2, 10-5, 10-9, 10-13

J

$J directive 5-43
Java 10-1
jump instructions (assembler) 13-3

K

.kof files 2-2

L

labels 4-1, 4-4, 4-18
  assembler 13-2
-$LE- compiler switch 9-12
Length function 5-11, 5-18, 5-20
less-than *See* Symbols
libraries *See* DLLs or dynamically loadable libraries
library (directive) 4-17
library (reserved word) 9-3
Library search path 3-6
line-feed 4-5
-$LN- compiler switch 9-12
LoadLibrary function 9-2
local (directive) 9-4
local directive (Linux only) 9-4
local identifiers 4-28
local variables 5-38, 6-10
  memory management 11-2
LocalAlloc 11-1
locales 5-13
logical operators 4-8
long strings 4-9, 5-10, 5-12
  files and 5-25
  in dynamically loadable libraries 9-8
  memory management 11-2, 11-6
  records and 5-23
LongBool type 5-6, 11-3
Longint type 5-4, 11-3
Longword type 5-4, 11-3
loop statements 4-20, 4-25
Low function 5-3, 5-4, 5-12, 5-18, 5-20, 6-16
-$LU- compiler switch 9-12

M

$M directive 7-4, 7-6
main form 2-5
$MAXSTACKSIZE directive 11-2
members *See* sets
members, of classes 7-1
  interfaces 10-2
  visibility 7-4
memory 4-1, 5-2, 5-25, 5-26, 5-31, 5-38, 7-14
  dynamically loadable libraries 9-6
  heap 5-39
  management 11-1 to 11-12
  overlays (in records) 5-23
  shared memory manager 9-8
memory references (assembler) 13-12
message (directive) 7-15
  interfaces 10-7
message dispatching 7-16
message handlers 7-15
  inherited 7-16
  overriding 7-16
Message property 7-32
Messages unit 7-15
metaclasses 7-23
method directives
  order 7-8
method pointers 4-12, 5-29
method resolution clauses 10-5, 10-6
methods 7-1, 7-2, 7-8 to 7-17
  abstract 7-12
  Automation 7-6, 10-12
  binding 7-10
  calling conventions 12-3
  class methods 7-1, 7-25
  constructors 7-13, 12-4
  destructors 7-14, 12-4
  dispatch interface 10-11
  dispatching calls 7-11
  dual-interface 6-5
  dynamic 7-10
  implementation 7-8
  overloading 7-12
  overriding 7-10, 7-11, 10-6
  pointers 4-12, 5-29
  publishing 7-5
  static 7-10
  virtual 7-6, 7-10
$MINSTACKSIZE directive 11-2
minus *See* Symbols
mod 4-6
modules *See* units
multibyte character sets 5-13
  string-handling routines 8-7
multidimensional arrays 5-18, 5-20, 5-43
multiple unit references 3-7
multiplication 4-6
multithreaded applications 5-39
  dynamically loadable libraries 9-7
mutually dependent classes 7-7
mutually dependent units 3-7

N

name (directive) 6-7, 6-8, 9-5
named parameters 10-12
names
  See also identifiers
  exported routines 9-5
  functions 6-3, 6-4
  identifiers 4-16
  packages 9-10
  programs 3-1, 3-2
  units 3-3, 3-6
naming conflicts 3-6, 4-29
near (directive) 6-5
negation 4-7
  bitwse 4-8
nested conditionals 4-23
nested exceptions 7-31
nested routines 5-29, 6-11
New procedure 5-19, 5-27, 5-39, 7-4, 9-8, 11-1, 11-2
nil 5-27, 5-31, 5-40, 11-5
nodefault specifier 7-6, 7-17, 7-21
not 4-6, 4-7
Null (variants) 5-31, 5-33
null character 5-13, 11-6, 11-7, 11-10

null string 4-4
null-terminated strings 5-13 to 5-16, 5-27, 11-6, 11-7
  mixing with Pascal strings 5-15
  standard routines 8-6, 8-7
numerals 4-1, 4-4
  as labels 4-4, 4-19
  assembler 13-9
  type 5-41, 6-8

O

object files
  calling routines in 6-7
Object Inspector 7-5
Object Inspector (Delphi) 7-5
object interfaces *See* interfaces, COM, CORBA
object types 7-4
objects 4-21, 7-1
  See also classes
  'of object' 5-29
  comparison 4-11
  files and 5-25
  memory 11-10
of (reserved word) 4-24, 5-17, 5-19, 5-25, 5-29, 6-15, 6-17, 7-23
of object (method pointers) 5-29
OleVariant 5-34
OleVariant type 5-28, 5-34
on (reserved word) 7-29
opcodes (assembler) 13-2
open array constructors 6-17, 6-20
open array parameters 6-15, 6-20
  and dynamic arrays 6-16
Open function 8-4, 8-5
OpenString 6-15
operands 4-6
operators 4-6 to 4-13
  assembler 13-15
  class 7-24
  precedence 4-12, 7-25
or 4-7, 4-8
Ord function 5-3, 5-4
order of method directives 7-8
ordinal types 5-2 to 5-9
ordinality 5-2
  enumerated types 5-6, 5-7
out (output) parameters 6-12, 6-13, 6-19
out (reserved word) 6-11, 6-12, 6-13
outer block 4-29
OutlineError 7-32
output (program parameter) 3-2
output *See* file I/O
Output variable 8-3
overloaded methods 7-12
  access specifiers 7-13, 7-18
  publishing 7-5
overloaded procedures and functions 6-6, 6-8
  default parameters 6-10, 6-19
  dynamically loadable libraries 9-5
  forward declarations 6-9
overriding interface implementations 10-6
overriding methods 7-10, 10-6
  hiding and 7-11
overriding properties 7-22
  access specifiers and 7-22
  Automation 7-6
  hiding and 7-22

P

$P directive 6-15
package files 2-2, 2-3, 9-8, 9-9, 9-11
packages 9-8 to 9-12
  compiler directives 9-12
  compiler switches 9-12
  compiling 9-11
  declaring 9-9
  loading dynamically 9-9
  loading statically 9-9
  thread variables 9-10
  uses clause and 9-9
packed (reserved word) 5-17, 11-8
packed arrays 4-5, 4-9, 5-18
packed records 11-8
packed strings 5-18
  comparison 4-11
pairs of symbols 4-2
PAnsiChar type 5-13, 5-27
PAnsiString type 5-28
parameters 5-29, 6-2, 6-3, 6-11 to 6-19
  See also overloaded procedures and functions
  actual 6-19
  array 6-12, 6-15
  array property indexes 7-19
  Automation method calls 10-12
  calling conventions 6-5
  constant 6-13, 12-1
  default 6-18 to 6-19, 10-12
  file 6-12
  formal 6-19
  names 10-12
  open array 6-15
  output (out) 6-13
  overloading and 6-6, 6-8, 6-9
  parameter list 6-11
  passing 12-1
  positional 10-12
  program control 12-1
  properties as 7-17
  registers 6-5, 12-2
  short strings 6-15
  typed 6-12
  untyped 6-14, 6-19
  value 6-12, 12-1
  variable (var) 6-12, 12-1
  variable number 6-6
  variant open arrray 6-17
parentheses *See* Symbols
partial evaluation 4-7
partitioning
  application 9-8
.pas files 2-3, 3-1, 3-3, 3-7
pascal (calling convention) 6-4, 12-2
  constructors and destructors 12-4
  Self 12-4
PByteArray type 5-28
PChar type 4-5, 4-9, 5-13, 5-14, 5-15, 5-27, 5-45
  comparison 4-11
PCurrency type 5-28
PDouble type 5-28
period *See* Symbols
PExtended type 5-28
PGUID 10-3
PInteger type 5-28
platform (directive) 4-17
plus *See* Symbols
Pointer type 5-25, 5-26, 5-27, 11-5
pointers 5-25 to 5-28
  arithmetic 4-9
  character 5-27
  constants 5-44
  files and 5-25
  functions 4-12, 5-29
  in var parameters 6-13
  in variant open array parameters 6-17
  long strings 5-16

  memory 11-5
  method pointers 5-29
  nil 5-27, 11-5
  null-terminated strings 5-14, 5-16
  operators 4-9
  overview 5-26
  Pointer type 4-12, 11-5
  pointer types 4-12, 5-26, 5-27 to 5-28, 11-5
  procedural types 4-12, 5-28 to 5-31
  records and 5-23
  standard types 5-27, 5-28
  variants and 5-31
POleVariant type 5-28
polymorphism 7-9, 7-11, 7-14
positional parameters 10-12
pound *See* Symbols
precedence of operators 4-12, 7-25
Pred function 5-3, 5-4
predecessor 5-2
predefined types 5-1
private class members 7-4, 7-5
procedural constants 5-44
procedural types 4-15, 5-28 to 5-31
  calling dynamically loadable libraries 9-2
  calling routines with 5-30
  compatibility 5-29
  default parameters 6-18
  in assignments 5-30
  memory 11-10
procedure pointers 4-12, 5-29
procedures 3-4, 6-1 to 6-20
  assembler 13-16
  calling externally 6-6
  declaring 6-2, 6-6
  nested 5-29, 6-11
  overloading 6-6, 6-8
  pointers 4-12, 5-29
  procedure calls 4-18, 6-1, 6-2, 6-19 to 6-20
program (reserved word) 3-2
program control 6-19, 12-1 to 12-6
programs 2-1 to 2-5, 3-1 to 3-9
  examples 2-3 to 2-5
  syntax 3-1 to 3-3
project files 2-2, 3-1, 3-2, 3-6
Project Manager 2-1
project options files 2-2
projects 2-5, 3-6
properties 7-1, 7-17 to 7-23
  access specifiers 7-17
  array 7-5, 7-19
  as parameters 7-17
  declaring 7-17, 7-19
  default 7-20, 10-2
  interfaces 10-4
  overriding 7-6, 7-22
  read-only 7-19
  record 7-5
  write-only 7-19
protected class members 7-4, 7-5
prototypes 6-1
PShortString type 5-28
PSingle type 5-28
PString type 5-28
PTextBuf type 5-28
Ptr function 5-27
public class members 7-4, 7-5
public identifiers (interface section) 3-4
published class members 7-4, 7-5
  $M directive 7-6
  restrictions 7-5
PVariant type 5-28
PVarRec type 5-28
PWideChar type 5-13, 5-14, 5-27
PWideString type 5-28
PWordArray type 5-28

Q

qualified identifiers 3-7, 4-2, 4-29, 5-22
  in typecasts 4-15
  pointers 5-27
  with Self 7-9
querying (interfaces) 10-10
QueryInterface method 10-2, 10-5, 10-10
quotation marks *See* Symbols
quoted strings 4-4, 5-45
  assembler 13-9
QWORD type (assembler) 13-15

R

raise (reserved word) 4-20, 7-27, 7-29, 7-30
range-checking 5-4, 5-5, 5-9
Read procedure 8-2, 8-3, 8-4, 8-5, 8-6
read specifier 7-6, 7-17
  array properties 7-19
  index specifier and 7-20
  object interfaces 10-1, 10-4, 10-7
  overloading 7-13, 7-18
Readln procedure 8-5, 8-6
readonly (directive) 10-2, 10-11
read-only properties 7-19
real (floating-point) operators 4-6
Real type 5-10
real types 5-9, 11-4
  comparison 4-11
  conversion 4-15
  publishing 7-5
Real48 type 5-9, 5-10, 7-5, 11-4
$REALCOMPATIBILITY directive 5-10
ReallocMem procedure 5-39, 11-1
records 4-21, 5-21 to 5-24
  constants 5-43
  in properties 7-5
  memory 11-8
  record types 5-21
  scope 4-29, 5-22
  variant parts 5-23 to 5-24
  variants and 5-31
recursive procedure and function calls 6-1, 6-4
reference-counting 5-12, 10-9, 11-6, 11-7
register (calling convention) 6-4, 7-6, 7-13, 7-14, 12-2
  constructors and destructors 12-4
  dynamically loadable libraries 9-4
  interfaces 10-3, 10-7
  Self 12-4
registers 6-5, 12-2, 12-3
  assembler 13-2, 13-10, 13-12, 13-17
  storing sets 11-7
reintroduce (directive) 7-12
relational operators 4-11
_Release method 10-2, 10-5, 10-9
relocatable expressions (assembler) 13-13
Rename procedure 8-6
repeat statements 4-20, 4-25
requires clause 9-8, 9-9, 9-10
.RES files 2-2, 3-2
reserved words 4-1, 4-2, 4-3

  See also directives
  assembler 13-7
  list 4-3
Reset procedure 8-2, 8-4, 8-5, 8-6
resident (directive) 9-5
resource files 2-2, 2-3, 3-2
resource strings 5-42
resourcestring (reserved word) 5-42
Result variable 6-3, 6-4
RET instruction 13-3
return type (functions) 6-3, 6-4
return value (functions) 6-3, 6-4
  constructors 7-13
Rewrite procedure 8-2, 8-4, 8-5, 8-6
Round function 5-4
routines 6-1 to 6-20
  See also functions, procedures
  exporting 9-5
  standard 8-1 to 8-10
RTTI 7-5, 7-12, 7-21
$RUNONLY directive 9-12
runtime binding *See* dynamic methods, virtual methods
runtime packages 9-8
runtime type information *See* RTTI

S

$S directive 11-2
safecall (calling convention) 6-4, 12-2
  constructors and destructors 12-4
  dual interfaces 10-13
  interfaces 10-3
  Self 12-4
scope 4-27 to 4-29
  classes 7-3
  records 5-22
  type identifiers 5-37
Seek procedure 8-2
SeekEof function 8-5
SeekEoln function 8-5
Self 7-9
  calling conventions 12-3
  class methods 7-25
semicolon *See* Symbols
separators 4-1, 4-5
SetLength procedure 5-11, 5-16, 5-19, 5-20, 6-16
SetMemoryManager procedure 11-2
sets
  empty 5-17
  memory 11-7
  operators 4-10
  publishing 7-5
  set constructors 4-13
  set types 5-17
  variants and 5-31
SetString procedure 5-16
setter *See* write specifier
shared object files 2-3, 9-1
  dynamically loadable 9-2
  exceptions 9-7
  importing functions 6-7
ShareMem unit 9-8
shift-left (bitwise operator) 4-8
shift-right (bitwise operator) 4-8
shl 4-8
short strings 5-3, 5-10, 5-12
short-circuit evaluation 4-7
Shortint type 5-4, 11-3
ShortString type 5-10, 5-12, 5-28, 11-5
  parameters 6-15
  variant arrays and 5-34
ShowException procedure 7-32
shr 4-8
sign
  in typecasts 4-15
  numerals 4-4
simple statements 4-17
simple types 5-2
Single type 5-9, 11-4
16-bit applications (backward compatibility) 6-5
SizeOf function 5-2, 5-5, 6-16
slash *See* Symbols
Smallint type 5-4, 11-3
.so files 9-1
source-code files 2-1
spaces 4-1
special symbols 4-1, 4-2
stack size 11-2
standard routines 8-1 to 8-10
  null-terminated strings 8-6, 8-7
  wide-character strings 8-7
statements 4-1, 4-17 to 4-27, 4-28, 6-1
static arrays 5-18, 11-7
  variants and 5-31
static methods 7-10
statically loaded libraries 9-1
stdcall (calling convention) 6-4, 12-2
  constructors and destructors 12-4
  interfaces 10-3
  Self 12-4
  shared libraries 9-4
storage specifiers 7-21
  array properties and 7-22
stored specifier 7-6, 7-17, 7-21
Str procedure 8-6
StrAlloc function 5-39
StrDispose procedure 5-39
streaming (data) 5-2, 7-5
string
  See also character sets
  comparison 4-11, 5-11
  constants 4-4, 5-45, 13-9
  handling *See also* standard routines, null-terminated strings
  in variant open array parameters 6-17
  indexes 4-14
  literals 4-4, 5-45
  memory management 11-5, 11-6
  null-terminated 5-13 to 5-16, 5-27
  operators 4-9, 5-15
  parameters 6-15
  types 5-10 to 5-16
  variant arrays 5-34
  variants 5-31
  wide 5-13, 8-7, 11-2
string (reserved word) 5-11
StringToWideChar function 8-7
strong typing 5-1
StrToInt64 function 5-4
StrToInt64Def function 5-4
structured statements 4-20
structured types 5-16
  files and 5-25
  records and 5-23
  variants and 5-31
structures 5-21
StrUpper function 5-15
subrange types 4-7, 4-24, 5-8
subset operator 4-10
subtraction 4-6
  pointers 4-9
Succ function 5-3, 5-4
successor 5-2
superset operator 4-10
symbol pairs 4-2
symbols 4-1, 4-2

  See also special symbols *and Symbols on page I-1*
  assembler 13-10
syntax
  descriptions 1-2
  formal A-1 to A-7
System unit 3-1, 3-5, 5-28, 5-31, 5-33, 6-17, 7-3, 7-28, 8-1, 8-7, 10-2, 10-3, 10-5, 10-10, 11-11
  dynamically loadable libraries 9-6, 9-7
  memory management 11-2
  modifying 8-1
  scope 4-29
  uses clause and 8-1
SysUtils unit 3-5, 5-28, 6-11, 6-17, 7-26, 7-27, 7-28, 7-32
  dynamically loadable libraries 9-8
  uses clause and 8-1

T

$T directive 4-12
tag (records) 5-23
TAggregatedObject 10-7
TBYTE type (assembler) 13-15
TByteArray type 5-28
TClass 7-3, 7-23, 7-24
TCollection 7-23
  Add method 7-9
TCollectionItem 7-24
TDateTime 5-33
text files 8-2, 8-3
Text type 5-25, 8-3
text-file device drivers 8-4
TextFile type 5-25
TGUID 10-3
then (reserved word) 4-22
thread variables 5-39
  in packages 9-10
threadvar 5-39
TInterfacedObject 10-2, 10-5
to (reserved word) 4-26
TObject 7-3, 7-16, 7-24
tokens 4-1
TPersistent 7-6
True 5-6, 11-3
true constants 5-40
Trunc function 5-4
try...except statements 4-20, 7-28
try...finally statements 4-20, 7-31
TTextBuf type 5-28
TTextRec type 5-28
TVarData 5-31, 11-11
TVarRec 5-28
TVarRec type 6-17
TWordArray 5-28
type identifiers 5-2
Type Library editor 10-3
typecasts 4-14 to 4-16, 7-8
  checked 7-25, 10-10
  enumerated types 5-8
  in constant declarations 5-40
  interface 10-10
  untyped parameters 6-14
  variants 5-32
type-checking (objects) 7-24
types 5-1 to 5-37
  array 5-18 to 5-21, 11-7
  assembler 13-13
  assignment-compatibility 5-36
  automatable 7-6, 10-11
  Boolean 5-5, 11-3
  built-in 5-1
  character 5-5, 11-3
  class 7-1, 7-2, 7-4, 7-6, 7-7, 7-8, 7-17, 11-10
  classification 5-1
  class-reference 7-23, 11-11
  compatibility 5-17, 5-29, 5-36
  constants 5-40
  declared 5-1
  declaring 5-37
  dispatch interface 10-10
  enumerated 5-6 to 5-8, 11-3
  exception 7-27
  file 5-25, 11-9
  fundamental 5-1
  generic 5-1
  integer 5-3, 11-3
  interface 10-1 to 10-4
  internal data formats 11-2 to 11-12
  object 7-4
  ordinal 5-2 to 5-9
  pointer 5-27 to 5-28
  predefined 5-1
  procedural 5-28 to 5-31, 11-10
  real 5-9, 11-4
  record 5-21 to 5-24, 11-8
  scope 5-37
  set 5-17, 11-7
  simple 5-2
  string 5-10 to 5-16, 11-5, 11-6
  structured 5-16
  subrange 5-8
  type identity 5-35
  user-defined 5-1
  variant 5-31 to 5-35
typographical conventions 1-2

U

UCS-2 5-13
UCS-4 5-13
unary operators 4-6
Unassigned (variants) 5-31, 5-33
underscores 4-2
Unicode 5-5, 5-13
union (sets) 4-10
UniqueString procedure 5-16
unit files 3-1, 3-3
  case-sensitivity 4-2
units 2-1, 3-1 to 3-9
  scope 4-29
  syntax 3-3 to 3-9
until (reserved word) 4-25
untyped files 5-25, 8-2, 8-4
untyped parameters 6-14
UpCase function 5-11
uses clause 2-1, 3-1, 3-2, 3-4, 3-5 to 3-9
  interface section 3-7
  ShareMem 9-8
  syntax 3-6
  System unit and 8-1
  SysUtils unit and 8-1

V

Val procedure 8-6
value parameters 6-12, 6-19, 12-1
  open array constructors 6-20
value typecasts 4-14
var (reserved word) 5-38, 6-11, 6-12, 12-1
varargs (directive) 6-6
VarArrayCreate function 5-34
VarArrayDimCount function 5-34
VarArrayHighBound function 5-34
VarArrayLock function 5-34, 10-12
VarArrayLowBound function 5-34
VarArrayOf function 5-34
VarArrayRedim function 5-34
VarArrayRef function 5-34
VarArrayUnlock procedure 5-34, 10-12

VarAsType function 5-32
VarCast procedure 5-32
variable (var) parameters 6-12, 6-19, 12-1
variable parameters 6-19
variable typecasts 4-14, 4-15
variables 4-6, 5-37 to 5-40
  absolute addresses 5-39
  declaring 5-38
  dynamic 5-39
  file 8-2
  from dynamically loadable libraries 9-1
  global 5-38, 10-9
  heap-allocated 5-39
  initializing 5-38
  local 5-38, 6-10
  memory management 11-2
  thread 5-39
variant arrays 5-31, 5-34
variant open array parameters 6-17, 6-20
variant parts (records) 5-23 to 5-24
variants 5-31 to 5-35, 11-12
  and Automation 11-12
  complete evaluation 4-8
  conversions 5-31, 5-32 to 5-33, 5-34
  files and 5-25
  freeing 5-40
  initializing 5-31, 5-38
  interfaces and 10-9, 10-10
  memory management 11-2, 11-11
  OleVariant 5-34
  operators 4-6, 5-33
  records and 5-23
  short-circuit evaluation 4-8
  typecasts 5-32
  variant arrays 5-34
  variant arrays and strings 5-34
  Variant type 5-28, 5-31
  variant types 5-31 to 5-35
varOleString constant 5-34
varString constant 5-34
VarType function 5-31
varTypeMask constant 5-31
VCL 1-2
virgule *See* Symbols
virtual method table 11-10
virtual methods 7-10
  Automation 7-6
  constructors 7-14
  overloading 7-12
VirtualAlloc function 11-1
VirtualFree function 11-1
visibility (class members) 7-4
  interfaces 10-2
Visual Component Library *See* VCL
VMT 11-10
VMTOFFSET 13-6

W

$WARNINGS directive 4-17
$WEAKPACKAGEUNIT directive 9-12
while statements 4-20, 4-25, 4-26
wide characters and strings 5-13
  memory management 11-2
  standard routines 8-7
WideChar type 4-9, 5-5, 5-11, 5-13, 5-27, 11-3
WideCharLenToString function 8-7
WideCharToString function 8-7
WideString type 5-10, 5-13, 5-28
  memory management 11-6
Windows 7-16
  memory management 11-1, 11-2
  messages 7-15
  variants and 11-11
Windows unit 9-2
with statements 4-20, 4-21, 5-22
Word type 5-4, 11-3
  assembler 13-14
WordBool type 5-6, 11-3
wrap-around (ordinal types) 5-4, 5-5
Write procedure 8-2, 8-3, 8-4, 8-5, 8-6
write procedures 5-3
write specifier 7-6, 7-17
  array properties 7-19
  index specifier and 7-20
  object interfaces 10-1, 10-4
  overloading 7-13, 7-18
Writeln procedure 2-4, 8-5, 8-6
writeonly (directive) 10-2, 10-11
write-only properties 7-19

X

$X directive 4-5, 4-18, 5-14, 6-1, 6-4
.xfm files 2-2, 2-7, 7-5
xor 4-7, 4-8

Z

-$Z- compiler switch 9-12
$Z directive 11-3